

**soft-rayed** \ 'sɒf-'træd\ *adj* 1 of a *n* : having soft articulated rays  
**2** : SOFT-FINNED  
**soft rot** *n* : a mushy, watery, or slimy decay of plants or their parts caused by bacteria or fungi  
**soft scale** *n* : a scale insect more or less active in all stages  
**soft sell** *n* : the use of suggestion or persuasion in selling rather than aggressive pressure  
**soft-shell** \ 'sɒf(t)-, shel\ or **soft-shelled** \-'sheld\ *adj* : having a soft or fragile shell esp. as a result of recent shedding  
**soft-shelled turtle** \,sɒf(t)-, shel(d)-\ *n* : any of numerous fiercely voracious aquatic turtles (family Trionychidae) with a flat shell covered with soft leathery skin instead of with horny plates  
**soft-shoe** \ 'sɒf(t)-'shu\ *adj* : of or relating to tap dancing done in soft-soled shoes without metal taps  
**soft soap** *n* 1 : a semifluid soap **2** : FLATTERY  
**soft-soap** \ 'sɒf(t)-'sɒp\ *vb* : to soothe or persuade with flattery or biarney — **soft-soap-er** \-'sɒ-'pə-\ *n*  
**soft-spoken** \-'spɒ-'kən\ *adj* : speaking softly : having a mild or gentle voice : SUAVE  
**soft-ware** \ 'sɒf-, twa(ə)r, -, twe(ə)r\ *n* : the features of an apparatus that are not hardware (the programs for a computer are ~)  
**soft wheat** *n* : a wheat with soft starchy kernels high in starch but usu. low in gluten  
**soft-wood** \ 'sɒf-, twʊd\ *n* 1 : the wood of a coniferous tree including both soft and hard woods **2** : a tree that yields softwood  
**softwood** *adj* 1 : having or made of softwood **2** : consisting of immature still pliable tissue (~ cuttings for propagating plants)  
**soft-wood-ed** \ 'sɒf-'twʊd-əd\ *adj* 1 : having soft wood that is easy to work or finish **2** : SOFTWOOD 1  
**softy or soft-ie** \ 'sɒf-tē\ *n* ['soft] 1 : an excessively sentimental or susceptible person **2** : a weak, effeminate, or foolish person  
**Sog-di-an** \ 'säg-dē-ən\ *n* [L *Sogdiani*, pl., fr. pl. of *sogdianus adj.*, *Sogdian*, fr. OPers *Sughuda Sogdiana*] 1 : a native or inhabitant of Sogdiana **2** : an Iranian language of the Sogdians — **Sogdian** *adj*  
**sog-gi-ly** \ 'säg-ə-lē, 'sɒg-\ *adv* : in a soggy manner  
**sog-gi-ness** \-ē-nəs\ *n* : the quality or state of being soggy  
**sog-gy** \ 'säg-ē, 'sɒg-\ *adj* [E dial. *sog* (to soak)] 1 : saturated or heavy with water or moisture : as **a** : WATERLOGGED, SOAKED (a ~ lawn) **b** : SODDEN (~ bread) **2** : heavily dull (~ prose)  
**soi-di-sant** \,swä-dē-'zä\ *adj* [F, lit., saying oneself] : SELF-Styled, SO-CALLED — usu. used disparagingly (a *soi-disant* artist)  
**soi-gné or soi-gnée** \swän-'yā\ *adj* [F, fr. pp. of *soigner* to take care of, fr. ML *soniare*] 1 : elegantly maintained : MODISH (a ~ restaurant) **2** : WELL-GROOMED, SLEEK  
**soil** \ 'sɔɪ(ə)\ *vb* [ME *soilen*, fr. OF *soiller* to wallow, soil, fr. *soil* pigsty, prob. fr. L *suile*, fr. *sus* pig — more at *sow*] *vt* 1 : to stain or defile morally : CORRUPT, POLLUTE **2** : to make unclean esp. superficially : DIRTY **3** : to blacken or besmirch (as a person's reputation) : SULLY, DISGRACE ~ *vi* 1 : to become stained or dirty  
**soil** *n* 1 **a** : SOILAGE 1, STAIN **b** : matter that causes staining or soiling **2** : something that soils or pollutes : as **a** : FERTILE SOIL, CLAY, etc. **b** : SEWAGE **c** : DUNG, EXCREMENT  
**soil** *n* [ME, fr. AF, fr. L *sollum* seat; prob. akin to L *sedere* to sit — more at *sit*] 1 : firm land : EARTH **2** : the upper layer of earth that may be dug or plowed : *specif* : the loose surface material of



**varian**  
**data machines**



# **varian software handbook**



**varian data machines**  
2722 michelson drive  
irvine/california/92664





# **Contents**

**Introduction**

**DAS Assemblers Reference Manual**

**Binary Load/Dump Program (BLD II) Reference Manual**

**Debugging Program (AID II) Reference Manual**

**Source Program Editor (EDIT) Reference Manual**

**Mathematical Package Reference Manual**

**FORTRAN IV Reference Manual**

**BASIC Reference Manual**

**Report Program Generator IV (RPG IV) Reference Manual**

**Master Operating System (MOS) Reference Manual**



# Introduction





# Introduction

This is your copy of the Varian Software Handbook.

Varian offers the computer user a comprehensive selection of mainframes, peripherals, and system configurations to do the job.

To back up this hardware capability, Varian also offers a choice of efficient, field proven software packages, human-engineered for simplified programming and operation.

These powerful and well-documented software packages can often save the user weeks or months of tedious system programming chores. With operating software available, the user can begin work at once on his particular application, concentrating all of his efforts on the problem to be solved.

The purpose of this Software Handbook is to bring together, in one convenient volume, the programming manuals for nine of the most important Varian 620 software systems (a tenth, VORTEX, is published in a separate volume).

## Assemblers

The selection of the appropriate software system is a function of both the job to be done and the hardware configuration.

If the principal objective is to write the most efficient programs possible, in terms of both speed and memory storage, the preferred route would be to write the programs in symbolic assembly language and to convert this into binary machine language through the use of a DAS Assembler program.

Three types of DAS assemblers are available. One is designed for minimum systems with only 4K of memory. The second is for systems with 8K of memory or more. The third, DAS MR, is a macro assembler which forms an integral part of the MOS and VORTEX operating systems (see below).

The use of the assemblers requires an intimate knowledge of the Varian 620 architecture and instruction set since the assembly-language symbolic statements are generally converted to machine-language instructions on a one-for-one basis.

## **introduction**

### **Language Processors**

Where the power and convenience of a high-level language are desirable, the user has a choice of FORTRAN IV, BASIC, or RPG IV.

With these language processors, several machine-language instructions may be represented by a single source statement. Because the ratio between source-statements and machine code instructions may be as high as ten-to-one, the number of statements that must be written by the programmer is considerably reduced.

The simplest, easiest-to-use language is BASIC. With only a few hours of instruction, an operator can program a Varian 620 computer and put it to useful work. The minimum configuration for BASIC is an 8K computer with the hardware multiply/divide option.

FORTRAN IV is a more efficient language, especially applicable to scientific applications. The FORTRAN IV language is fully compatible with the American National Standard Institute (ANSI) FORTRAN. The package is available either as a stand-alone version requiring only 8K of memory, or as an integral part of the MOS and VORTEX operating systems.

A special package for business applications is provided by the RPG IV (Report Program Generator) hardware/software system. The minimum hardware for an RPG IV system is a 4K computer, a card reader, a card punch, and a line printer. Application programs written in RPG IV language are far more efficient than the equivalent programs in COBOL. Built into the language are such functions as automatic scaling of numerical data, sequence-checking, control-break testing, and auditing of input records and fields.

To provide additional capabilities and simplify language enhancement, all three Varian high-level languages, FORTRAN IV, BASIC, and RPG IV, include special "CALL" statements for communication with assembly language subroutines. Through the use of these statements, the user can easily reference his own special applications and routines, while retaining the convenience of a high-level language processor.

### **Operating Systems**

Two comprehensive software operating systems are available from Varian for use with Varian 620 computers: MOS and VORTEX.

Both systems incorporate a full repertoire of utility programs, as well as the DAS MR and FORTRAN IV language processors. They are designed to enhance system utility by automating certain routine computer system operations.

MOS (Master Operating System) is specifically designed for batch-processing applications with data and programs stored on a magnetic tape or on a rotating memory. The minimum configuration for a MOS system is a Varian 620 computer with 8K of memory (12K if FORTRAN IV is to be supported), an ASR teletype, and a mass-storage unit such as a disc, drum, or magnetic tape.

**VORTEX (Varian Omnitask Real-Time Executive)** is a multi-programming system with special features designed for real-time applications. A number of different tasks may be stored in the main memory or on a rotating memory device. These tasks are scheduled by a resident executive program that gives highest priority to real-time "foreground" programs. Lower priority "background" programs are executed during the idle-time intervals embedded in most real-time operations. The effect is to give the system the utility of two computers for the price of one.

**VORTEX** also increases the efficiency of any installation in which the computer is required to operate on a number of different programs in sequence. The user simply establishes the priority of the jobs to be executed; **VORTEX** automatically schedules and runs the programs without further operator intervention. Details on **VORTEX** are given in a separate volume.

### **Hardware Configuration**

The minimum hardware prerequisites for each of the software systems described in this Handbook are given in Figure 1-1.



|                                                                   | 620 Family CPU<br>4K Memory<br>TTY                      | 620 Family CPU<br>8K Memory<br>TTY                             | 620 Family CPU<br>8K Memory<br>TTY<br>BIC (if disc<br>or drum)<br>Mass Storage | 620 Family CPU<br>12K Memory<br>TTY<br>BIC (if disc<br>or drum)<br>Mass Storage | 620/f CPU<br>16K Memory<br>TTY, Mem Protect<br>BIC, Power Fail/Restart<br>Disc Memory, Set<br>Optional Inst. Set<br>PIM, Real Time Clock<br>Paper Tape or Card RDR,<br>or Mag. Tape |
|-------------------------------------------------------------------|---------------------------------------------------------|----------------------------------------------------------------|--------------------------------------------------------------------------------|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SOFTWARE                                                          |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| DAS 4A ASSEMBLER<br>BLD II, AID II, EDIT,<br>Mathematical Package |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| RPG - IV                                                          | Also requires card reader, card punch, and line printer |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| DAS 8A ASSEMBLER                                                  |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| FORTRAN IV                                                        |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| BASIC                                                             |                                                         | Also requires hardware multiply/divide and extended addressing |                                                                                |                                                                                 |                                                                                                                                                                                     |
| M.O.S.                                                            |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| M.O.S. WITH<br>FORTRAN IV                                         |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |
| EXTENDED BASIC                                                    |                                                         |                                                                |                                                                                | Also requires hardware multiply/divide and extended addressing                  |                                                                                                                                                                                     |
| VORTEX                                                            |                                                         |                                                                |                                                                                |                                                                                 |                                                                                                                                                                                     |

Figure 1-1. Minimum Hardware Requirements for Software Systems

# **DAS Assemblers Reference Manual**





## SECTION I – DAS ASSEMBLERS

The Varian 620 assembler language (DAS) specifies instructions, addresses, address modifications, and constants in a straightforward and meaningful manner. Instruction mnemonics replace the usual numerical codes. Memory addresses can be referenced symbolically, thus offering a flexibility not attainable with absolute addressing. Constants can be used without conversion to binary or octal values. For ease in checkout and documentation, comments can be added between symbolic statements, or appended to the statements themselves.

DAS coding reduces machine language bookkeeping without compromising full utilization of the computer.

DAS 4A operates in a Varian 620 minimum-configuration system comprising the computer, 4K memory, and an on-line Teletype (TTY). DAS 8A requires 8K memory. Both can use and enhance additional system components, e.g., magnetic or paper tape equipment, card equipment, additional memory, line printer, etc. DAS MR operates under control of the Varian Master Operating System (MOS).

DAS translates symbolically coded instructions and data (source program) into binary instructions and data (object program).

### The DAS Character Set

The DAS character set comprises:

- a. Alphabetical characters

ABCDEFGHIJKLMNOPQRSTUVWXYZ\$

- b. Numerical characters

0123456789

- c. TTY characters

CR (carriage return)

LF (line feed)

**d. Special characters**

|   |                     |
|---|---------------------|
| + | (plus sign)         |
| - | (minus sign)        |
| * | (asterisk)          |
| / | (slash)             |
| . | (period)            |
|   | (blank)             |
| @ | (at sign)           |
| [ | (left bracket)      |
| ] | (right bracket)     |
| < | (less-than)         |
| > | (greater-than)      |
| ↑ | (up arrow)          |
| ← | (left arrow)        |
| ? | (question mark)     |
| = | (equal sign)        |
| , | (comma)             |
| ' | (prime)             |
| ( | (left parenthesis)  |
| ) | (right parenthesis) |
| \ | (back slash)        |
| ! | (exclamation point) |
| " | (Quotation mark)    |
| # | (pound sign)        |
| % | (percent sign)      |
| & | (ampersand)         |
| : | (colon)             |
| ; | (semicolon)         |

**Statement Format**

Each DAS source statement comprises a combination of label, operation, variable, and comment fields depending on the requirements of the instruction or directive (page 1-32).

**Label Field**

Symbols in the label field comprise one to four alphanumeric characters for the DAS 4A and DAS 8A assemblers and from one to six such characters for the DAS MR assembler. In any case, the first character is alphabetical. While only the given number of characters are recognized by the assembler, additional characters can be added.

Symbols are usually attached only to those source statements referenced elsewhere in the program, but this is not mandatory.

**Operation Field**

This field contains mnemonics for computer instructions and assembler directives. An asterisk following the mnemonic indicates indirect addressing. The mnemonics can be redefined with OPSY directives.

## Variable Field

The purpose of this field varies according to the requirements of the operation. The variable field can consist of a symbol, a constant, or an expression combining symbols and constants. Expressions in DAS are similar to arithmetic expressions except that parentheses do not appear. The variable field can contain the following operators: + (addition), - (subtraction), \* (multiplication), / (division).

Arithmetic operations always involve all 16 bits of the words. Operations are performed from left to right, with multiplication and division being performed before addition and subtraction. Thus,  $A + B/C * D$  in DAS is equivalent to  $A + (B/C) * D$  in conventional notation.

Use of the asterisk in the variable field gives access to the current value of the location counter. This asterisk is translated as the current value of the location counter, i.e., \* + 1 means the current value of the location counter plus one.

DAS constants are described in the following sections, wherein unsigned numbers are considered positive. DAS recognizes decimal and octal integers; floating-point numbers; alpha, address, and indirect constants; and literals.

A **decimal integer** is a signed or unsigned string of decimal digits, the first of which is not zero.

Examples:

1      29      -3      -9000

An **octal integer** is a signed or unsigned string of octal digits, the first of which is zero.

Examples:

07      -044      + 022745

A **floating-point number** has the form:

$\pm$  integer.fraction E  $\pm$  exponent

where the right parenthesis, at least one digit, and the decimal point are always present. Other items in the format are optional.

Examples:

)375.64E + 7                      )9.E - 2,.1E + 12  
)-4. + 20

An **alpha constant** is a string of characters within primes (''). Internally, it is represented in eight-bit ASCII code. Each memory word can contain two characters, where blanks are

## DAS assemblers

also recognized as characters. In the DAS 8A assembler, an alpha constant can be a term in an arithmetic expression. However, if more than one word is generated by the constant, only the last word generated is subject to arithmetic manipulation.

Examples:

'A'\*04000      'AB' + 1      'ABCD' + 011

where, in the last example, two words are generated and 011 added to the second word.

An **address constant** is a symbol, number, or expression enclosed in parentheses. It generates a 15-bit direct address (bit 15 = 0).

Examples:

(a + 2)      (3)      (a)

where a is an address symbol whose value is taken from the symbol table.

An **indirect address constant** is an address constant within parenthesis and followed by an asterisk. It generates a 15-bit indirect address (bit 15 = 1).

Examples:

(a + 2)\*      (3)\*      (a)\*

where a is an address symbol whose value is taken from the symbol table.

A **literal** comprises an expression preceded by an equal sign.

Examples:

= 3      = + 3      = .044      = (a + 2)\*  
= 'A',      = 'GO'

Literals permit reference to a constant in the variable field where the assembler generates the data and assigns memory addresses. Even where a literal is used many times, only one memory address is assigned for each literal. Note that the expressions reflect the values assigned to the symbols rather than the contents of the memory addresses referenced by the symbols.

### Comments Field

The comments field is separated from the variable field by at least one blank. It is used for any desired comments and is ignored by the assemblers.

## Modes of Symbols and Expressions

Each symbol or expression has one of the modes external (E), common (C), relative (R), or absolute (A), assigned by the assembler. The mode of an expression is determined by the mode of the symbols in the expression.

The mode of a symbol is determined by the following rules:

- a. If the symbol is defined by an EXT directive, the mode is E.
- b. If the symbol is defined by a COMN directive, the mode is C.
- c. If the symbol is a symbol in a program, or if \* is the current location counter value, the mode is R.
- d. If the symbol is a numerical constant, the mode is A.
- e. If the symbol is defined by EQU, SET, or similar directive, the mode of the symbol is that of the variable field expression in the directive.

The mode of an expression is determined by the following rules:

- a. If the expression contains any mode E or C symbol, the expression is mode E.
- b. If the expression contains only mode A symbols, the expression is mode A.
- c. If the expression contains mode A and R symbols, the mode of the expression is R if there is an odd number of mode R symbols. Otherwise, the mode of the expression is A.

The following restrictions apply only to DAS MR and to FORTRAN-compatible output assembly within DAS 8A.

- a. No expression can contain symbols of both modes E and C.
- b. A mode E expression comprises a single mode E symbol.
- c. No mode E, C, or R expression can multiply or divide a mode E or C symbol.
- d. No expression can add or subtract a mode C and a mode R symbol, or a mode E and a mode R symbol.
- e. No expression can add two or more mode E, C, or R symbols.
- f. A mode A symbol can be added to or subtracted from a mode C or R symbol.

Figure 1-1 illustrates the above rules.



|      |      |               |                                |
|------|------|---------------|--------------------------------|
| EEEE | EXT  |               | EEEE defined as type E         |
| CCCC | COMN | 6             | CCCC defined as type C         |
| RTN  | ENTR |               | RTN is type R, a symbol        |
| TBL  | BSS  | 50            | TBL is type R                  |
| ABL  | BSS  | 'A' + 5       | ABL is type R                  |
| LENG | EQU  | *-TBL         | LENG is type A, length of area |
|      |      | CALL          | EEEE, TBL, LENG                |
|      | LDA  | * + 6         | OK, relative forward           |
|      | LDA  | CCCC + 6      | Illegal, one word inst,        |
|      |      |               | not R or A                     |
|      | LDXI | CCCC + 6      | OK, two word instruction       |
|      | LDA  | 0,1           | Get CCCC + 6 to A, legal       |
|      | DATA | EEEE + 4      | Illegal, value not zero        |
|      | DATA | CCCC + 4      | Legal                          |
|      | DATA | CCCC + LENG   | Legal                          |
|      | DATA | TBL + LENG -5 | Legal, mode is R               |

Figure 1-1. Manipulation of Expression and Symbol Modes

## Assembler Instructions

DAS assemblers recognize all Varian 620 instructions, even when the system lacks the hardware for execution. The programmer is responsible for knowing the instructions applicable to his system.

All DAS assembler instructions have the format:

```
label field
operation field
variable field
```

where the label field is optional and contains a symbol when used, the operation field contains the instruction mnemonic, and the variable field contains one, two, or three expressions. Where there are two or three expressions, they are separated by commas.

Assembler instructions are of five types described below and in figure 1-2. Figure 1-3 lists the instruction mnemonics by type.

### Addressing of Single Word

If an address lower than 2048 is specified without indirect addressing, the assembler generates an instruction with direct addressing.

If indexing is specified, the assembler generates an indexed instruction.

If the address specified is 1 to 512 words (inclusive) beyond the current instruction, the assembler generates an instruction with relative addressing.

If indirect addressing with a data address lower than 512 is specified, the assembler generates an instruction with indirect addressing and the specified address.

In all other cases, including indirect addressing with an address higher than 511, the assembler generates an instruction with indirect addressing and the specified address, stores the address in a table, and inserts the storage address in the referencing instruction. Duplicate values in the table are discarded.

### Instruction Types

Figure 1-2 : gives the characteristics of the types of assembler instructions and figure 1-3 lists the instructions by type.

A **type 1 instruction** occupies one computer word and is memory-addressing.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field in parentheses.

|                                   | Type 1 | Type 2 | Type 3 | Type 4 | Type 5 |
|-----------------------------------|--------|--------|--------|--------|--------|
| Words generated                   | 1      | 2      | 2      | 1      | 2      |
| Expressions in the Variable field | 1 to 2 | 1      | 2      | 1      | 1 to 3 |
| Memory addressing                 | Yes    | Yes    | Yes    | No     | Yes    |
| Indirect addressing               | Yes    | Yes    | Yes    | No     | Yes    |
| Indexing                          | Yes    | No     | No     | No     | Yes    |
| Condition micro-coding            | No     | No     | Yes    | Yes    | No     |

Figure 1-2. Characteristics of Assembler Instruction Types

| Type 1 | Type 2 |       | Type 3 | Type 4 |      | Type 5 |
|--------|--------|-------|--------|--------|------|--------|
| ADD    | ADDI   | JS3N  | BT     | AOFA   | LRLB | ADDE   |
| ANA    | ANAI   | JS3NM | IME    | AOFB   | LSRA | ANAE   |
| DIV    | DIVI   | JXNZ  | JIF    | AOFX   | LSRB | DIVE   |
| ERA    | ERAI   | JXNZM | JIFM   | ASLA   | MERG | ERAE   |
| INR    | INRI   | JXZ   | JMIF   | ASLB   | NOP  | IJMP   |
| LDA    | JAN    | JXZM  | OME    | ASRA   | OAB  | INRE   |
| LDB    | JANM   | LDAI  | SEN    | ASRB   | OAR  | JSR    |
| LDX    | JANZ   | LDBI  | XIF    | CIA    | OBR  | LDAE   |
| MUL    | JANZM  | LDXI  |        | CIAB   | ROF  | LDBE   |
| ORA    | JAP    | MULI  |        | CIB    | SEL  | LDXE   |
| STA    | JAPM   | ORAI  |        | COMP   | SEL2 | MULE   |
| STB    | JAZ    | STAI  |        | CPA    | SOF  | ORAE   |
| STX    | JAZM   | STBI  |        | CPB    | SOFA | \$RE   |
| SUB    | JBNZ   | STXI  |        | CPX    | SOFB | STAE   |
|        | JBNZM  | SUBI  |        | DAR    | SOFX | STBE   |
|        | JBZ    | XAN   |        | DBR    | TAB  | STXE   |
|        | JBZM   | XANZ  |        | DECR   | TAX  | SUBE   |
|        | JMP    | XAP   |        | DXR    | TBA  |        |
|        | JMPM   | XAZ   |        | EXC    | TBX  |        |
|        | JOE    | XBNZ  |        | EXC2   | TSA  |        |
|        | JOEM   | XBZ   |        | HLT    | TXA  |        |
|        | JOEN   | XEC   |        | IAR    | TXB  |        |
|        | JOENM  | XOF   |        | IBR    | TZA  |        |
|        | JSS1   | XOFN  |        | INA    | TZB  |        |
|        | JSS2   | XS1   |        | INAB   | TZX  |        |
|        | JSS3   | XS1N  |        | INB    | ZERO |        |
|        | JS1M   | XS2   |        | INCR   |      |        |
|        | JS1N   | XS2N  |        | IXR    |      |        |
|        | JS1NM  | XS3   |        | LASL   |      |        |
|        | JS2M   | XS3N  |        | LASR   |      |        |
|        | JS2NM  | XXNZ  |        | LLRL   |      |        |
|        | JS3M   | XXZ   |        | LLSR   |      |        |
|        |        |       |        | LRLA   |      |        |

Figure 1-3. Assembler Instructions by Type

Examples:

|      |               |
|------|---------------|
| LDA* | expression    |
| LDA  | (expression)* |

Indexing is specified by a variable field having two expressions. The first is the indexing increment and is less than 512. The second specifies the indexing register where one is the X register and two is the B register.

Example:

|     |        |
|-----|--------|
| LDA | 0300,1 |
|-----|--------|

loads the A register with the contents of the memory address specified by the sum of the contents of the X register plus 0300. Thus, if the X register contains 0200, the operand for this instruction is in memory address 0500 (0300 + 0200).

A **type 2 instruction** occupies two consecutive computer words and is memory-addressing. The second word is the address of a jump, jump-and-mark, or execution instruction; or the second word of an immediate instruction. The variable field contains only one expression.

Indirect addressing is specified as with an assembler type 1 instruction. There is no indexing with assembler type 2 instructions.

A **type 3 instruction** occupies two consecutive computer words and is memory-addressing. It differs from an assembler type 2 instruction in that the variable field contains two expressions.

For the JIF, JIFM, JMIF, and XIF instructions, the first expression specifies the condition(s) required for the jump, jump-and-mark, or execution.

Example:

|     |           |
|-----|-----------|
| JIF | 0222,ALFA |
|-----|-----------|

takes the next instruction from address ALFA if the A register contains a positive number (0002), the B register contains zero (0020), and SENSE switch 2 is set (0200) since  $0222 = 0002 + 0020 + 0200$ .

For the IME, and OME instructions, the first expression specifies the I/O device address. For SEN, the first expression specifies the function and I/O device.

For the BT instruction, the first expression specifies the register and bit to be tested.

Indirect addressing is specified by an asterisk after the mnemonic or after a variable field expression in parentheses.

Examples:

|      |              |
|------|--------------|
| JIF* | 0222,ALFA    |
| JIF  | 0222,(ALFA)* |

**Note:**

Do not indirect address IME or OME.

A **type 4 instruction** occupies one computer word and does not address memory.

For ZERO, DECR, MERG, COMP, INCR, and the register modification instructions, the assembler generates an instruction as specified by the value in the variable field.

A **type 5 instruction** occupies two consecutive computer words and is memory-addressing.

Indirect addressing is specified by an asterisk after the mnemonic or after the first expression in the variable field when the expression is enclosed in parentheses.

Examples:

|       |                           |
|-------|---------------------------|
| LDAE* | expression                |
| LDAE  | (expression)*             |
| LDAE  | (expression)*, expression |

Pre-indexing is specified by two expressions in the variable field, where the first expression is the data address and the second specifies the indexing register: one for the X register, two for the B register.

Example:

|      |         |
|------|---------|
| LDAE | 01300,2 |
|------|---------|

loads the A register with the contents of the memory address specified by the sum of the contents of the B register plus 01300. Thus, if the B register contains 0400, the operand for this instruction is in memory address 01700 (01300 + 0400).

Post-indexing is specified by three expressions in the variable field, where the first expression is the data address, the second specifies the indexing register as in pre-indexing, and the third is logically ORed with the instruction word.

Example:

```
LDAE      ADDR,2,0200
```

loads the A register extended and postindexed with the B register. The assembler does not check the validity of the third expression, but its value cannot exceed 0200. Pre-indexing is not applicable to IJMP, JSR, or SRE.

## Assembler Directives

Directives are instructions to the assembler. They are divided into the following groups:

- a. Symbol definition directives
- b. Instruction definition directives
- c. Location counter control directives
- d. Data definition directives
- e. Memory reservation directives
- f. Conditional assembly directives
- g. Assembler control directives
- h. Subroutine control directives
- i. List and punch control directives
- j. DAS 8A interface directive to stand-alone FORTRAN
- k. Program link directives
- l. MOS I/O control directives
- m. Macro definition directives

Directives have four fields:

- a. The label field, which contains a symbol. The label field is usually optional. However, some directives require it by the nature of their operations, e.g., EQU, SET, BEGI, and FORM.
- b. The operation field, which contains the directive mnemonic.
- c. The variable field, whose contents vary with the directive. Any symbols in this field that are not previously defined cause error flags to be output.

- d. The comment field (optional) for programming convenience.

In the following descriptions of the individual directives, the format

```

label field
operation field
variable field

```

is used, with the optional comment field being understood to follow the variable field when used. In cases where the variable field contains more than one item or expression, these are always separated by commas.

Figure 1-4 shows the directives recognized by each DAS assembler, but does not include MOS I/O control directives accepted by the assembler.

## Symbol Definition Directives

These directives assign arbitrary values to symbols in the symbol table. This table is a list of symbols in the source program. For each symbol in the table, there is a corresponding value, usually an address.

### **EQU** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

```

symbol      EQU      expression

```

It places the symbol in the assembler's symbol table and assigns it the value of the expression. If the symbol has already been entered in the symbol table, error message DD is output and the value in the table is replaced by that of the expression. Any symbol used as the variable field expression must have been defined previously. The label field symbol is mandatory.

### **SET** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

```

symbol      SET      expression

```

It is the same as EQU except that there is no error message output if the symbol has already been entered in the symbol table.

### **MAX** (DAS 8A)

This directive has the format

```

symbol      MAX      expression(s)

```



| Directive | DAS 4A | DAS 8A | DAS MR |
|-----------|--------|--------|--------|
| BEGI      | Yes    | Yes    | No     |
| BES       | Yes    | Yes    | Yes    |
| BSS       | Yes    | Yes    | Yes    |
| CALL      | Yes    | Yes    | Yes    |
| COMN      | No     | Yes    | Yes    |
| CONT      | No     | Yes    | Yes    |
| DATA      | Yes    | Yes    | Yes    |
| DETL      | No     | Yes    | Yes    |
| DUP       | No     | Yes    | Yes    |
| EJEC      | No     | Yes    | Yes    |
| EMAC      | No     | No     | Yes    |
| END       | Yes    | Yes    | Yes    |
| ENTR      | Yes    | Yes    | Yes    |
| EQU       | Yes    | Yes    | Yes    |
| EXT       | No     | Yes    | Yes    |
| FORM      | No     | Yes    | Yes    |
| FORT      | No     | Yes    | No     |
| GOTO      | No     | Yes    | Yes    |
| IFF       | No     | Yes    | Yes    |
| IFT       | No     | Yes    | Yes    |
| LIST      | No     | Yes    | No     |
| LOC       | Yes    | Yes    | Yes    |
| MAC       | No     | No     | Yes    |
| MAX       | No     | Yes    | No     |
| MIN       | No     | Yes    | No     |
| MORE      | Yes    | Yes    | No     |
| MZE       | Yes    | Yes    | Yes    |
| NAME      | No     | Yes    | Yes    |
| NLIS      | No     | Yes    | No     |
| NPUN      | No     | Yes    | No     |
| NULL      | No     | Yes    | Yes    |
| OPSY      | No     | Yes    | Yes    |

Figure 1-4. Directives Recognized by DAS Assemblers (1 of 2)

| Directive | DAS 4A | DAS 8A | DAS MR |
|-----------|--------|--------|--------|
| ORG       | Yes    | Yes    | Yes    |
| PUNC      | No     | Yes    | No     |
| PZE       | Yes    | Yes    | Yes    |
| READ      | No     | Yes    | No     |
| RETU *    | Yes    | Yes    | Yes    |
| SET       | Yes    | Yes    | Yes    |
| SMRY      | No     | Yes    | Yes    |
| SPAC      | No     | Yes    | Yes    |
| USE       | No     | Yes    | No     |

Figure 1-4. Directives Recognized by DAS Assemblers (2 of 2)

It assigns to the symbol the largest algebraic value found among the expressions. Any symbol used as a variable field expression must have been defined previously. The label field symbol is mandatory. Use SET to redefine the symbol.

### MIN (DAS 8A)

This directive has the format

|        |     |               |
|--------|-----|---------------|
| symbol | MIN | expression(s) |
|--------|-----|---------------|

It is the same as MAX except that the symbol is assigned the smallest algebraic value found among the expressions.

### Instruction-Definition Directive

This directive redefines a standard instruction mnemonic.

### OPSY (DAS 8A, DAS MR)

This directive has the format

|        |      |          |
|--------|------|----------|
| symbol | OPSY | mnemonic |
|--------|------|----------|

It makes the symbol a mnemonic with the mnemonic. The variable field must be a standard mnemonic.

Example:

|     |      |      |
|-----|------|------|
| CLA | OPSY | LDA  |
|     | CLA  | BETA |

### Location Counter Control Directives

These directives control the location counter(s). DAS 8A has multiple location counters and directives to modify or preset their values. Figure 1-5 lists the five standard DAS 8A location counter symbols and their uses. They need not be created by the user. However, up to eight other location counters in addition to those provided can be created, thus providing complex relocatable and overlay programs within a single assembly.

There are no user-created location counters at the beginning of an assembly. The assembler uses three location counters for location assignment. Thus, IAOR and LTOR are always in use, as is a third counter used to assign locations to generated instructions and generated data (except literals and indirect pointers). This is done by the blank location counter until USE specifies another.

In a straightforward program using only one location counter, complete control over the counter is maintained by ORG and LOC.

| Counter | Initial Value | Use                                                                                                      |
|---------|---------------|----------------------------------------------------------------------------------------------------------|
| SYOR    | 00000         | Controls assignment of memory to all system parameters                                                   |
| IAOR    | 00200         | Controls assignment of memory to indirect pointers                                                       |
| LTOR    | 01000         | Controls assignment of memory to literals                                                                |
| COMN    | 02000         | Controls assignment of memory within an interface area common to two or more programs                    |
| blank   | 04000*        | Used initially and normally by the assembler for memory assignments until/ unless overridden by USE; = 0 |

\* Zero for DAS 4A Assembler

**Figure 1-5. Standard DAS 8A Location Counters**

**ORG (DAS 4A, DAS 8A, DAS MR)**

This directive has the format

symbol          ORG          expression

where the symbol is optional. It sets the location counter currently in use to the value of the expression. If a symbol is present in the label field, it is set to the value of the expression. Any symbol used as the variable field expression must have been defined previously.

**LOC (DAS 4A, DAS 8A, DAS MR)**

This directive has the format

symbol          LOC          expression

where the symbol is optional. It is used if the data and instructions following the LOC address are to be moved to the LOC address by the object program before execution, i.e., to keep a block of data or instructions undisturbed by assembly. LOC causes data or instructions following it to be generated as if there had been a ORG to change the current location counter value. However, this value is not actually changed. Any symbol used as a variable field expression must have been defined previously. LOC cannot be used in a relocatable program.

**BEGI (DAS 4A, DAS 8A, DAS MR)**

This directive has the format

symbol          BEGI          expression

where the symbol is optional. It creates a new location counter or it redefines the value of any location counter before the counter has been used. BEGI gives the new or redefined location counter the value of the expression, but has no effect on the current location counter. BEGI cannot redefine the value of any location counter that has been used for location assignment. Any symbol used as a variable field expression must have been defined previously. BEGI is also used to change the starting location of the IAOR or LTOR counters.

**USE (DAS 4A, DAS 8A)**

This directive has the format

blank          USE          xxxx

where xxxx is a blank, COMN, SYOR, or a user-created location counter label. It uses the location counter xxxx to assign locations to data and instructions (except literals and indirect pointers). If xxxx is PREV, the previously used location counter is recalled with the restriction that only the last-used counter can be so recalled.

### Data Definition Directives

These directives control the sign and assignment of data words. In the descriptions, item refers to a data item, which can be an expression or a direct or indirect address.

#### **DATA** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol      DATA      expression(s)

where the symbol is optional. DATA assigns the symbol to the memory address of the first generated word. In the absence of a symbol, an unlabeled block of data is generated.

#### **PZE** (DAS 4A, DAS 8A, DAS MR)

This directive (PZE = plus zero) has the format

symbol      PZE      expression(s)

It is similar to DATA except that the sign bits of data words are always zeros (plus).

#### **MZE** (DAS 4A, DAS 8A, DAS MR)

This directive (MZE = minus zero) has the format

symbol      MZE      expression(s)

It is similar to DATA except that the sign bits of data words are always ones (minus).

#### **FORM** (DAS 8A, DAS MR)

This directive has the format

symbol      FORM      expression(s)

where the symbol is mandatory and the terms are absolute terms or expressions. The symbol is the name of the FORM, which specifies the format of a bit configuration of a data word. The terms specify the length in bits of each field in the generated data word, where the sum of the term values is from one to the number of bits in the computer word. FORM is ignored if there are any errors in the variable field, except that an error is flagged when a term cannot be represented in the number of bits specified when FORM is applied (by placing its name in the operation field of a symbolic source statement) to another statement. The FORM can be redefined.

## DAS assemblers

Example:

|      |      |         |
|------|------|---------|
| BYTE | FORM | 8,8     |
| BCD  | FORM | 4,4,4,4 |
| PTAB | FORM | 1,2,3,4 |

given the FORM definition

|     |      |       |
|-----|------|-------|
| ABC | FORM | 6,2,8 |
|-----|------|-------|

and the FORM reference,

|     |           |
|-----|-----------|
| ABC | 2*3,1,'A' |
|-----|-----------|

the generated binary data word is:

|   |     |     |     |     |     |
|---|-----|-----|-----|-----|-----|
| 0 | 001 | 100 | 111 | 000 | 001 |
|---|-----|-----|-----|-----|-----|

### Memory Reservation Directives

These directives control the reservation of memory addresses and areas.

**BSS** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

|        |     |            |
|--------|-----|------------|
| symbol | BSS | expression |
|--------|-----|------------|

where the symbol is optional. It increases the value of the current location counter by the amount indicated by the expression. If there is a symbol, it is assigned the value of the counter prior to such increase. The location counter always points to the next available word. An expression value of zero will result in the symbol being assigned to the next address available.

**BES** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

|        |     |            |
|--------|-----|------------|
| symbol | BES | expression |
|--------|-----|------------|

It is similar to BSS except that if there is a symbol, it is assigned to the address one less than the incremented location counter. An expression value of zero will result in the symbol being assigned to the last address used.

**DUP** (DAS 8A, DAS MR)

This directive has the formats

|       |     |     |
|-------|-----|-----|
| blank | DUP | n   |
| blank | DUP | n,m |

It duplicates source statement(s) following the DUP. The first format duplicates the next source statement  $n$  times. The second format duplicates the next  $m$  source statements  $n$  times, where  $m \leq 3$  and  $n \leq 32,767$ .

### Conditional Assembly Directives

These directives assemble portions of the program according to the conditions specified in the variable fields.

#### IFT (DAS 8A, DAS MR)

This directive (IFT = if true) has the format

blank          IFT          expression(s)

When a single expression is present in the variable field, the next symbolic source statement is assembled if the value of the expression is not zero. Otherwise, the next symbolic source statement is not assembled.

When two expressions are present in the variable field separated by two commas (i.e., A,,B), the next symbolic source statement is assembled if the value of the first expression is not equal to the value of the second expression. Otherwise the next symbolic source statement is not assembled.

When three expressions are present in the variable field separated by commas (i.e., A,B,C), the next symbolic source statement is assembled if the value of the first expression is less than the value of the second expression and the value of the second expression is less than or equal to the value of the third expression. Otherwise, the next symbolic source statement is not assembled.

Examples:

| Operation<br>Field | Variable<br>Field | The next source<br>statement is assembled |
|--------------------|-------------------|-------------------------------------------|
| IFT                | A,,B              | for $A \neq B$                            |
| IFT                | A,B,B             | for $A \leq B$                            |
| IFT                | 0,A,B             | for $A \leq B$ and $A > 0$                |
| IFT                | A                 | for $A \neq 0$                            |

#### IFF (DAS 8A, DAS MR)

This directive (IFF = if false) has the format

blank          IFF          expression(s)

When a single expression is present in the variable field, the next symbolic source statement is assembled if the value of the expression is zero. Otherwise, the next symbolic source statement is not assembled.



When two expressions are present in the variable field separated by two commas (i.e., A,,B), the next symbolic source statement is assembled if the value of the first expression is equal to the value of the second expression. Otherwise, the next symbolic source statement is not assembled.

When three expressions are present in the variable field separated by commas (i.e., A,B,C), the next symbolic source statement is assembled if the value of the first expression is greater than or equal to the value of the second expression or the value of the second expression is greater than the value of the third expression. Otherwise, the next symbolic source statement is not assembled.

Examples:

| Operation<br>Field | Variable<br>Field | The next source<br>statement is assembled |
|--------------------|-------------------|-------------------------------------------|
| IFF                | A,,B              | for $A = B$                               |
| IFF                | A,B,B             | for $A \geq B$                            |
| IFF                | A                 | for $A = 0$                               |
| IFF                | 0,A,B             | for $A > B$ or $A \leq 0$                 |

### **GOTO** (DAS 8A, DAS MR)

This directive has the formats

|       |      |         |       |      |          |
|-------|------|---------|-------|------|----------|
| blank | GOTO | symbol  | blank | GOTO | integer  |
| blank | GOTO | symbol, | blank | GOTO | integer, |

It skips more than one source statement and usually follows an IFF or IFT. All source statements following the GOTO up to, but not including the statement containing the symbol in its label field are deleted from the assembly process. GOTO cannot jump back to an earlier point in the program. If the first GOTO format is used, the skipped instructions are listed. If the second format (containing a comma after the symbol) is used, the listing of the skipped instructions is suppressed. This listing can also be suppressed by a SMRY directive.

### **CONT** (DAS 8A, DAS MR)

This directive has the format

|        |      |       |
|--------|------|-------|
| symbol | CONT | blank |
|--------|------|-------|

It provides a target for a previous GOTO.

The symbol is not entered in the assembler symbol table.

### **NULL** (DAS 8A, DAS MR)

This directive has the format

symbol      NULL      blank

It provides a target for a previous GOTO with the symbol entered in the symbol table. NULL has the same effect as BSS with a blank variable field.

### Assembler Control Directives

These directives signal the end or continuance of an assembly.

#### **MORE** (DAS 4A, DAS 8A)

This directive has the format

blank      MORE      blank

It halts assembly to allow additional source statements to be put in the input device. Assembly resumes when the RUN or START button on the computer control panel is pressed. MORE is never listed.

#### **END** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

blank      END      expression

It is the last source statement in the program. The expression is the execution address of the program after it has been loaded into the computer. A blank expression yields an execution address of 00000.

### Subroutine Control Directives

These directives create closed subroutines and control their use.

#### **ENTR** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol      ENTR      blank

where the symbol is the name of the subroutine called. ENTR generates a linkage word of zero in the object program.

#### **RETU\*** (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol      RETU\*      expression

## DAS assemblers

where the symbol is optional. It returns from a closed subroutine, generating an unconditional indirect jump to the address indicated by the value of the expression.

### CALL (DAS 4A, DAS 8A, DAS MR)

This directive has the format

symbol      CALL      subfield(s)

where the symbol is optional and the subfields comprise a required symbol, an optional parameter list, and an optional error return list. A symbol in the label field is entered in the symbol table and assigned the present value of the current location counter. The symbol required in the variable field is the name of a subroutine. The optional lists in the variable field comprise valid data items.

Example:

CALL      FUNC,X,Y + 1,(ERR),(GOOF)\*

produces a machine code identical to that obtained with

JMPM      FUNC  
DATA      X,Y + 1,(ERR),(GOOF)\*

### List and Punch Control Directives

These directives, which are operative only during the second pass of the assembler, control listing and punching during program assembly.

#### LIST (DAS 8A)

This directive has the format

blank      LIST      blank

It causes the assembler to produce a program listing. Initially, the assembler is in LIST condition.

#### NLIS (DAS 8A)

This directive has the format

blank      NLIS      blank

It suppresses further listing of the program.

#### SMRY (DAS 8A, DAS MR)

This directive has the format

blank          SMRY          blank

It suppresses the listing of source statements that have been skipped under control of conditional assembly directives. In DAS 8A, it also suppresses the listing of symbols on the first pass.

**DETL** (DAS 8A, DAS MR)

This directive has the format

blank          DETL          blank

It removes the effect of SMRY, i.e., causes listing of all source statements, including those skipped by conditional assembly directives. Initially, the assembler is in DETL condition.

**PUNC** (DAS 8A)

This directive has the format

blank          PUNC          blank

It causes the assembler to produce an object program. Initially, the assembler is in PUNC condition.

**NPUN** (DAS 8A)

This directive has the format

blank          NPUN          blank

It suppresses further production of paper tape punched with the object program.

**SPAC** (DAS 8A, DAS MR)

This directive has the format

blank          SPAC          blank

It causes the listing device to skip a line. SPAC is not listed.

**EJEC** (DAS 8A, DAS MR)

This directive has the format

blank          EJEC          blank

## DAS assemblers

It causes the listing device to move to the next top of form. EJEC is not listed.

### READ (DAS 8A)

This directive has the format

blank        READ        n,p

where n is the number of characters (20-80) from each source line to be processed by the assembler, and p is either 26 or 29 to indicate use of IBM 026 or 029 keypunch codes, respectively. Initially, the assembler processes 80 characters per line with 026 keypunch codes. If n is outside the range 20-80, the assembler resets the number of characters to 80 and outputs the error message SZ (size). If p is not 26 or 29, assembly stops with the A, B, X, and I registers equal to 026. To continue assembly, correct the card, reinsert it in the card reader, and press RUN on the computer control panel. If n is used without p, there is no comma. If p is used without n, it is preceded by a comma.

Examples:

|                   |                                                                           |
|-------------------|---------------------------------------------------------------------------|
| <b>READ 80,29</b> | Reads 80 columns of 029 codes on succeeding cards                         |
| <b>READ 72,26</b> | Reads 72 columns of 026 codes on succeeding cards                         |
| <b>READ ,29</b>   | Does not change number of columns read, but reads 029 on succeeding cards |
| <b>READ 80</b>    | Reads 80 columns but does not change the code read                        |

Unless there is an SZ error message, SMRY suppresses the listing of READ cards during the second pass.

### DAS 8A Interface Directive to FORTRAN

This directive has the format

blank        FORT        blank

Except for comments, this is the first line of an assembly whose output is compatible with the stand-alone FORTRAN relocatable loader. In such an assembly, all two-word instructions are legal, but one-word memory-addressing instructions are limited to relative forward addressing. An expression containing symbols defined with COMN or EXT directives must be the second word of two-word instructions, or part of a DATA, PZE, or MZE directive. Literals are forbidden. Immediate instructions are recommended. The symbols and expressions are subject to the mode restrictions indicated for FORTRAN-compatible output.

Example:

|        |   |      |      |           |
|--------|---|------|------|-----------|
| 000000 | R |      | FOR  |           |
| 000017 | R |      | NAME | \$PE      |
| 000000 | E | \$SE | EXT  |           |
| 000000 | E | \$QS | EXT  |           |
| 000000 | E | \$QE | EXT  |           |
|        |   |      |      |           |
| 000000 |   |      | STX  | \$PE + 7  |
| 000001 |   |      | LDX  | \$PE + 4  |
| 000002 |   |      | STA  | \$PE + 9  |
| 000003 |   |      | STB  | \$PE + 10 |
| 000004 |   |      | LDA  | 0,1       |
| 000005 |   |      | LDX  | \$PE + 7  |
| 000006 |   |      | JMPM | \$QS      |
| 000007 | E |      |      |           |
| 000010 | R |      | DATA | \$PE + 7  |
| 000011 |   |      | LDA  | \$PE + 9  |
| 000012 |   |      | LDB  | \$PE + 10 |
| 000013 |   |      | JMPM | \$QE      |
|        |   |      |      |           |
| 000014 | E |      |      |           |
| 000015 | R |      | DATA | \$SP + 7  |
| 000016 |   |      | JMP  | 0         |
| 000017 |   |      |      |           |
| 000017 |   |      | ORG  | *.1       |
| 000017 |   | \$PE | ENTR |           |
| 000020 |   |      | CALL | \$SE,1    |
| 000021 | E |      |      |           |
| 000022 |   |      |      |           |
| 000023 |   |      | DATA | 0         |
| 000024 |   |      | JMP  | *.20      |
| 000025 | R |      |      |           |
| 000026 |   |      | DATA | 0,0,0,0   |
| 000027 |   |      |      |           |
| 000030 |   |      |      |           |
| 000031 |   |      |      |           |
| 000000 |   |      | END  |           |

### Program Link Directives

These directives establish and control links among programs that have been assembled separately but are to be loaded and executed together.

**NAME** (DAS 8A, DAS MR)

## DAS assemblers

This directive has the format

blank            NAME

It establishes linkage definition points among separately assembled programs. The symbols can then be referenced by other programs. Each symbol also appears in the label field of a symbolic source statement in the body of the program. Undefined NAME symbols cause error messages to be output.

Examples:

```
NAME      A
NAME      A,B
NAME      EX,WHY,ZEE
```

**EXT** (DAS 8A, DAS MR)

This directive has the format

symbol            EXT            symbol(s)

where both the label field and variable field symbols are optional.

It establishes linkage definition points among separately assembled programs. The symbols declare each symbol not defined within the current program. Each symbol, in both the label and variable fields, is output to the loader with the address of the last reference to the symbol. If a symbol is not defined within the current program and not declared in an EXT, it is considered undefined and causes an error message output. If a symbol is declared in an EXT but not referenced within the current program, it is output to the loader for loading, but no linkage to this program is established. If a symbol is both defined in the program and declared to be external, the EXT declaration is ignored.

Examples:

```
          EXT      AY
BEG       EXT      BE,SEE
          EXT      DEE,EE,EF,GEE
```

**COMN** (DAS 8A, DAS MR)

This directive has the format

symbol            COMN            item

where item is an absolute expression, and symbol is optional. It defines an area in blank common for use at execution time. This allows an assembler program to reference the same blank common area as a FORTRAN program. The common area is cumulative

for each use of COMN, i.e., the first COMN is the base of blank common, the second COMN is the base of blank common plus the size of the first, etc.

Example:

|     |      |     |
|-----|------|-----|
| AAA | COMN | 3   |
|     | COMN | 6*2 |
| BBB | COMN | 9   |

## MOS I/O Control Directives

DAS MR accepts the MOS control directives listed below and explained in the Varian 620 Master Operating System Reference Manual:

|      |                                        |
|------|----------------------------------------|
| RBIN | Read binary record                     |
| RALF | Read alphanumeric record               |
| RBCD | Read binary-coded decimal (BCD) record |
| WBIN | Write binary record                    |
| WALF | Write alphanumeric record .            |
| WBCD | Write BCD record                       |
| WEOF | Write end of file                      |
| REW  | Rewind                                 |
| SKFF | Skip files forward                     |
| SKFR | Skip files reverse                     |
| SKRF | Skip records forward                   |
| SKRR | Skip records reverse                   |
| FUNC | Function                               |
| STAT | Status                                 |
| ION  | I/O logical unit number                |

## Macro Definition Directives

These directives begin and end macro definitions. The macro is the assembly equivalent of the execution subroutine. It is defined once and can then be called from the program. The macro is an algorithmic statement of a process that can vary according to the arguments supplied. It is assembled with the resultant data inserted into the program at each point of reference, whereas the subroutine executed during execution time appears but once in a program. Its definition comprises the statements between MAC and EMAC.

### MAC (DAS MR)

This directive has the format

|        |     |       |
|--------|-----|-------|
| symbol | MAC | blank |
|--------|-----|-------|

and introduces a macro definition. The symbol is the name of the macro.

### EMAC (DAS MR)



## DAS assemblers

This directive has the format

blank           EMAC           blank

and terminates the definition of a macro.

A macro is called by the appearance of its name in the operation field of a symbolic source statement. The variable field of this statement contains expression(s), P(1),P(2),...,P(n), that are evaluated and stored in a table within the assembler. The macro definition is then processed with the values in the table being substituted for the respective values of the expressions in the source statement variable field. For example, if the variable field of the symbolic source statement contains:

2,B,9 + 8, = 63

then within the generated macro, P(1) = 2, P(2) is the value of B, P(3) = 021, and P(4) is the address of the value 63. All terms and expressions within the macro-referencing symbolic source statement parameter list are evaluated prior to calling the macro.

If the label field of such a source statement contains a symbol, the symbol is assigned the value and relocatability of the location counter at the time the macro is called but before data generation.

A macro definition can contain references to machine instruction mnemonics or to assembler directives other than DUP. Macros can be nested within macros to a depth limited only by the available memory at assembly time.

Example:

- a. Definition of the macro:

```
SBR  MAC
      SEN      0200 + P(1),* + 3
      JMP      *.2
      EMAC
```

- b. Calling the macro:

```
SBR      031
```

- c. Expansion of the macro:

```
SEN      0231,* + 3
JMP      *.2
```

The parameter P(0) can also be referenced within the macro. P(0) is the first entry in the table formed by the assembler and contains the number of entries in that table, i.e., the

number of parameters on the call line. This example shows the output listing obtained by calling P(0):

|       |       |     |   |             |
|-------|-------|-----|---|-------------|
|       |       | 1   | A | MAC         |
|       |       | 2   |   | DATA P(0)   |
|       |       | 3   |   | EMAC        |
| 00001 | 00000 | A 4 |   | A           |
| 00002 | 00001 | A 5 |   | A 1         |
| 00003 | 00002 | A 6 |   | A 1,2       |
| 00004 | 00003 | A 7 |   | A 1,2,3     |
| 00005 | 00004 | A 8 |   | A 1,2,3,4   |
| 00006 | 00005 | A 9 |   | A 1,2,3,4,5 |
|       |       | 10  |   | END         |

Relocatability Rules

A relocatable program is one that has been assembled with its instruction locations assigned in such a manner that it can be loaded anywhere in memory and executed without problems. When such a program is loaded, the beginning memory address is specified, and a value (known as the relocation bias) is added to the addresses of subsequent relocatable instructions. The programs are usually assembled with a zero relocation bias on the first instruction.

The location counter contains the (relative) address of the instruction currently being executed. The location counter is absolute when it contains the actual address of the instruction during execution, and relocatable when it contains the relative address of the instruction during execution (the current address minus the address of the start of the program).

Symbols can be absolute or relocatable. Expressions, since they contain symbols, can be absolute or relocatable. Constants are always absolute.

Figure 1-6 shows, for each arithmetic operation, whether the result is absolute (abso), relocatable (relo), or illegal.

The loader can load a relocatable program in any area of memory and modify the addresses as it loads so that the resulting program executes correctly. Programs can contain absolute and relocatable addresses and absolute and relocatable data. At the

|       | A is abso,<br>B is abso, | A is abso,<br>B is relo, | A is relo,<br>B is abso, | A is relo,<br>B is relo, |
|-------|--------------------------|--------------------------|--------------------------|--------------------------|
| A + B | abso                     | relo                     | relo                     | illegal                  |
| A - B | abso                     | illegal                  | relo                     | abso                     |
| A*B   | abso                     | illegal                  | illegal                  | illegal                  |
| A/B   | abso                     | illegal                  | illegal                  | illegal                  |

Figure 1-6. Results of Arithmetic Operations

## **DAS assemblers**

beginning of each assembly, the location counter is set to zero, relocatable. It is incremented after every instruction word or data word generated by the assembler. It can be set by the ORG directive. If an ORG directive is encountered the location counter is made absolute if the corresponding expression is absolute, or relocatable if the corresponding expression is relocatable.

If a symbol is equated to the location counter, it is relocatable if the location counter is relocatable. Otherwise, the symbol is absolute.

## **Source Statement Formats**

### **Punched Card Format**

Punched cards used as input to the DAS assembler contain four fields: label, operation, variable, and comment.

The label field is in columns one through six. Its use is optional. It assigns a symbol, extended symbol, or target number to a symbolic source statement. Column 7 must be blank.

The operation field is in columns eight through 14. It contains a mnemonic for a machine instruction or an assembler directive. Indirect addressing is specified by an asterisk following the mnemonic. Column 15 must be blank.

The variable field begins in column 16 and ends with the first blank not contained in a character constant. Its use depends on the instruction or directive. The variable field contains zero or more subfields, but if two or more subfields are present, they are separated by commas.

The comment field fills the remainder of the card. If there is a blank variable field, the comment field begins in column 17. The assembler ignores this field, but its contents appear in the output listing.

An asterisk in column one indicates that the entire card contains a comment. The assembler ignores the card, but its contents appear in the output listing.

### **Paper Tape Format**

Paper tape used as input to the DAS assembler contains source statements of up to 80 characters each (not including the carriage return and line feed characters). Each statement contains four fields: label, operation, variable, and comment. The first three are separated by commas, and the comment field starts after the first variable field blank that is not part of a character constant. At the end of each statement is a carriage return (CR), followed by a line feed (LF).

The label field contains a symbol, extended symbol, or target number. If the first nonblank character in the statement is a comma, the label field is blank.

The operation field contains a mnemonic. An asterisk following the mnemonic specifies indirect addressing.

The variable field can be blank, or contain one or more subfields separated by commas. Each subfield can contain a constant or expression, depending on the instruction or directive, or it can be voided by using adjacent commas. The variable field terminates with a blank that is not part of a character, constant, or with a CR or LF.

The comment field fills the remainder of the statement from the terminating blank of the variable field to the next CR or LF. The assembler ignores this field, but its contents appear in the output listing.

If the first nonblank character of the statement is an asterisk, the entire statement is a comment. The assembler ignores the statement, but its contents appear in the output.

Note: The instruction field must immediately follow the first comma and the variable field must follow the second comma.

## Assembler Output Listing

### DAS Source/Object Listing

The listing can be obtained in whole or in part as the program is being assembled. The source (symbolic) program and the object (absolute) program are listed side-by-side on the listing device output paper. This device is either a Teletype or a line printer.

The listing is output according to the specifications given by the list and punch control directives in the assembly.

Error analysis during assembly causes the error messages described in the following section to be output on the line following the point of detection.

The following example illustrates the format of the output listing. In addition, on DAS MR listings, a line count appears. The addressing modes are: FORTRAN common reference = C, externally defined = E, indirect pointer = I, and absolute or relative = R.

Example:

| Location | Code   | Mode | Source Statement |
|----------|--------|------|------------------|
| 014000   |        |      | ORG 014000       |
| 014000   | 000000 |      | ABS ENTR         |
| 014001   | 001002 |      | JAP* ABS         |
| 014002   | 114000 | R    |                  |
| 014003   | 005211 |      | CPA              |
| 014004   | 001000 |      | JMP* ABS         |
| 014005   | 114000 | R    |                  |
|          | 000000 |      | END              |

### DAS Error Messages

The assembler checks syntax on both passes. During pass 1, detectable errors are listed. During pass 2, the following information is listed:

## DAS assemblers

- a. Error code
- b. Value of location counter
- c. Object code when the instruction is assembled

This code is suppressed by NLIS directives and list-suppression commas in GOTO directives.

The error message appears in the listing line following the statement found in error. Each line can hold up to four error messages. Figure 1-7 lists the error codes recognized by DAS, and their meanings.

## DAS 4A I/O Operations

Load the program into memory using the binary loader. Execute it by entering a positive, nonzero value in the A register during loading, or by clearing all registers and pressing RESET and RUN after the loading.

During execution, the program first determines the amount of memory required. It then stores in memory location 00003 a value one less than the lower limit of the binary load/dump. This is the highest address that the assembler section can use without destroying part of the binary load/dump.

### I/O Device Definition

The program then makes three requests for definitions of I/O devices:

#### ENTER DEVICE NAME FOR XX

where xx is one of the I/O function names SI (source input), LO (list output), or BO (binary output), respectively. Respond to each request in turn by typing the name of the desired device and a carriage return (CR). Figure 1-8 lists the acceptable device names in response to each request. If the default assignment is desired, merely press CR. If the device name is incorrect, the message

#### DEVICE NAME NOT VALID

is output and the request repeated.

Output of any line to the TTY can be terminated by pressing RUBOUT. The error correction feature can be used during specification of I/O devices to the program.

When I/O assignments are complete, the I/O section uses the binary loader to load the assembler section into memory.

To restart the I/O section before the assembler section is loaded, press STEP, clear all registers, and press RESET and RUN.

| Code | Meaning                                                                                                                                                                                                                                 |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| *AD  | An address expression is in error.                                                                                                                                                                                                      |
| *DC  | A decimal character appears in an octal constant.                                                                                                                                                                                       |
| *DD  | There is an invalid redefinition of a symbol or the location counter.                                                                                                                                                                   |
| *E   | The symbolic source statement is incorrectly formed.                                                                                                                                                                                    |
| *EX  | An expression contains an illegal construction.                                                                                                                                                                                         |
| *FA  | A floating point number contains a format error.                                                                                                                                                                                        |
| *IL  | The first non-blank character of the source statement is invalid; the statement is not processed.                                                                                                                                       |
| *NR  | No memory space available for the addition of an entry to the assembler's tables.                                                                                                                                                       |
| *NS  | No symbol in the label field of a SET, EQU, MAC or FORM directive statement. No symbol in the label or variable field of an OPSY directive statement. No symbol in the variable field of a NAME directive statement.                    |
| *OP  | The instruction field is undefined; two No Operation (NOP) instructions are generated in the object program. The remainder of the statement is not processed. Illegal nesting of DUP or MAC directive statements also cause this error. |
| *QQ  | Illegal use of prime (').                                                                                                                                                                                                               |
| *R   | A relocatable item was encountered in the place where an absolute item was expected.                                                                                                                                                    |
| *SE  | The symbol in the location field has a value during pass 2 that is different than the value used in pass 1.                                                                                                                             |
| *SY  | An expression contains an undefined symbol.                                                                                                                                                                                             |

Figure 1-7. DAS Error Codes (1 of 2)

|     |                                                                                                                                                                  |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| *SZ | The expression value is too large for the size of the subfield or a DUP statement specifies that more than three symbolic source statements are to be assembled. |
| *TF | Undefined or illegal index register specification.                                                                                                               |
| *UC | An undefined character in an arithmetic expression.                                                                                                              |
| *UD | Undefined symbol in variable field of USE directive.                                                                                                             |
| *XR | Address out of range for indexing specification.                                                                                                                 |
| * = | Illegal use of a literal.                                                                                                                                        |

**Figure 1-7. DAS Error Codes (2 of 2)**

| Assembly Function  | Device                                          | Default Assignment |
|--------------------|-------------------------------------------------|--------------------|
| SI (source input)  | TR (TTY paper tape reader)                      | TR                 |
|                    | TY (TTY keyboard)                               |                    |
|                    | PR (high-speed paper tape reader)               |                    |
|                    | CR (Model 620-22, 23, and 25 card readers)      |                    |
|                    | CR1 (Model 620A card reader, micro-verse logic) |                    |
| LO (list output)   | TY (TTY printer)                                | TY                 |
|                    | LP (Model 620-76 line printer)                  |                    |
|                    | LP1 (Model 620-75 line printer)                 |                    |
|                    | LP2 (Model 620-77 line printer)                 |                    |
| BO (binary output) | TP (TTY paper tape punch)                       | TP                 |
|                    | PP (high-speed paper tape punch)                |                    |
|                    | CP (Model 620-27 card punch)                    |                    |
|                    | CP1 (Model 620-26 card punch)                   |                    |

Figure 1-8. Acceptable I/O Devices



The following sections describe the subroutines applicable to each type of I/O function.

### Source Input I/O Subroutines

The following I/O devices can be source program input devices. The I/O subroutines to communicate with these devices are in the I/O section.

- a. 33/35 ASR TTY
- b. 33/35 ASR TTY paper tape reader
- c. High-speed paper tape reader
- d. Card reader

The **TY subroutine** communicates with the 33/35 ASR TTY. The (effective) calling sequence is:

|        |            |
|--------|------------|
| JMPM   | 0500       |
| DATA   | WORD COUNT |
| DATA   | ADDRESS    |
| Return |            |

TY inputs characters from the TTY keyboard until either a carriage return is input or the word count (two characters per word) specified in the calling sequence is reached.

As each character is input, it is verified as an ASCII character. If it is not, it is ignored. ASCII characters are output to the TTY page printer and are stored sequentially in memory beginning at the address specified in the calling sequence.

If the specified word count is 36 or larger, 36 words are input.

After the specific number of words or a carriage return is input, a line feed and a carriage return are output to the TTY page printer, signifying that input is complete.

To delete a line, while it is being input, type a backslash (press the shift key and the letter L simultaneously). A line feed and a carriage return are output to indicate that the line has been deleted. Characters entered before the backslash become blanks.

To delete a character just entered, type a backarrow. This is printed by the TTY page printer to indicate the deletion. As many backarrows as necessary can be entered. Each deletes one character (but not beyond the beginning of the line). Characters deleted change to blanks (in the data) but are printed with their accompanying backarrows on the page printer (each succeeding backarrow indicating deletion of the preceding character), e.g.:

IDEAX - indicates the X is deleted  
BOTHER - - indicates that R and E are deleted

The **TR subroutine** communicates with the 33/35 ASR TTY. The (effective) calling sequence is:

```

JMPM      0500

DATA      WORD COUNT
DATA      ADDRESS
Return

```

TR inputs characters from the TTY paper tape reader until either a carriage return is input or the word count (two characters per word) specified in the calling sequence is reached.

As each character is input, it is verified as an ASCII character. If it is not, it is ignored. ASCII characters are stored sequentially in memory beginning at the address specified in the calling sequence.

After the specified number of words has been input, characters are input but not stored in memory until there is a carriage return. When a carriage return is input, TR inputs two more paper-tape characters and stores them in memory locations 4 and 5. These two characters appear in the beginning of the next record when the next call is made to input a record from the TTY paper tape reader.

The **PR subroutine** communicates with the high-speed paper tape reader. The (effective) calling sequence is:

```

JMPM      0500
DATA      WORD COUNT
DATA      ADDRESS
Return

```

PR inputs characters from the high-speed paper tape reader until either a carriage return is input or the word count (two characters per word) specified in the calling sequence is reached.

As each character is input, it is verified as an ASCII character. If it is not, it is ignored. ASCII characters are stored sequentially in memory (two characters per computer word) beginning at the address specified in the calling sequence.

After the number of words specified in the calling sequence has been input, characters are input but not stored in memory until there is a carriage return. When a carriage return is input, PR terminates input from the high-speed paper tape reader.

The **CR subroutine** communicates with the card reader. The (effective) calling sequence is:

## DAS assemblers

|        |            |
|--------|------------|
| JMPM   | 0500       |
| DATA   | WORD COUNT |
| DATA   | ADDRESS    |
| Return |            |

CR inputs columns from the card reader and converts them to ASCII characters until the word count (two characters per word) specified in the calling sequence is reached. The ASCII characters are stored sequentially in memory beginning at the address specified in the calling sequence.

### Binary Output I/O Subroutines

The following I/O devices can be source program output devices. The I/O subroutines to communicate with these devices are in the I/O section.

- a. 33/35 ASR TTY paper tape punch
- b. High-speed paper tape punch

The **TP subroutine** communicates with the 33/35 ASR TTY. The (effective) calling sequence is:

|        |            |
|--------|------------|
| JMPM   | 01300      |
| DATA   | WORD COUNT |
| DATA   | ADDRESS    |
| Return |            |

TP outputs data to the TTY paper tape punch in a format compatible with the binary loader. The data are output from memory beginning at the specified address. The number of data words to be output is also specified in the calling sequence.

The binary loader requires three paper-tape frames for each data word. The most significant four bits are contained in the first frame; the next six bits in the second, and the least significant six bits in the third.

Each frame of paper tape can contain eight bits of information. Frame bit positions 1 through 6 contain the data. Frame bit position 7 contains a check bit that is the inverse of position 6. Frame bit position 8 is zero.

If the specified data word count is negative, 100 frames of paper tape are punched with frame bit position 8 punched and positions 1 through 6 not punched (leader/trailer). Any entry in the data address portion of the calling sequence is ignored.

The **PR subroutine** communicates with the high-speed paper tape punch. PP operates in the same manner as TP.

### List Output I/O Subroutines

The following devices can be list output devices. The subroutines to communicate with these devices are included in the I/O section.

- a. 33/35 ASR TTY page printer
- b. Line printer

The **TY subroutine** communicates with the 33/35 ASR TTY. The (effective) calling sequence is:

```
JMPM      01000
DATA      WORD COUNT
DATA      ADDRESS
Return
```

TY outputs ASCII characters to the TTY page printer. They are packed two per computer word and are output from memory beginning at the specified address.

If the specified word count is 36 or larger, 36 words are output.

When the specified number of words has been output, a line feed and a carriage return are output.

If the specified data word count is negative, nine line feeds and a carriage return are output to position the paper to the top of the next page. Any entry in the data address portion of the calling sequence is ignored.

The **LP subroutines** communicate with the line printer. The (effective) calling sequence is:

```
JMPM      01000
DATA      WORD COUNT
DATA      ADDRESS
Return
```

LP outputs ASCII characters to the line printer. The characters are packed two per computer word and are output from memory beginning at the specified address. If the specified word count is 66 or larger, 66 words are output.

Single-line slewing (paper movement) is supplied with each line output.

If the specified data word count is negative, the next line is printed at the top of the next page. Any entry in the data address portion of the calling sequence is ignored.

## Assembler Section

When control is given to the assembler by the binary loader, it halts with a one in the program counter (P register). The operator should set SENSE switch 1 ON if an assembly pass 1 is desired, or SENSE switch 1 OFF and 2 and 3 ON if an assembly pass 2 is desired.

## DAS assemblers

If an assembly pass 1 is selected, the operator should ready the source input I/O device (S1) with the symbolic source input statements and press RUN. The assembler proceeds to input the symbolic source statements and performs a pass 1 assembly. When the symbolic source END statement (denoting the completion of the pass 1 assembly process) is assembled, the assembler halts with a one in the program counter (P register) and a -1 in the A register. The operator should again select the desired assembly pass as described above.

If an assembly pass 2 is selected, the operator should ready (S1) with the symbolic source input statements, ready the binary output (BO) and list output (LO) I/O devices, and press RUN. The assembler proceeds to input the symbolic source statements and performs a pass 2 assembly outputting binary object loader text and an assembly listing. The binary output can be suppressed by setting SENSE switch 3 OFF. The assembly listing output can be suppressed by setting SENSE switch 2 OFF. Error diagnostics cannot be suppressed and are output on LO as they are detected during pass 2. It should be noted that SENSE switches 2 and 3 are only interrogated during pass 2 and have no effect during pass 1.

Pass 2 can be restarted or repeated without repeating pass 1.

END terminates both passes 1 and 2 by executing a HALT. After pass 1 is terminated, begin pass 2 by resetting the I/O devices, setting the SENSE switches as above, and pressing RUN. To obtain extra copies of the program, repeat pass 2 as desired. END is indicated by 017777 in the A register.

A MORE directive causes the computer to stop and wait until the input units are prepared and RUN is pressed. MORE is indicated by 0170017 in the A register.

Synchronization errors detected on pass 2 indicate that the value of a label on that pass does not agree with the value it had on pass 1. Such errors are due to source tape misreadings. They halt the assembly, as indicated by 0777 in the A register. To continue, press RUN. The assembler resets the location counter value to that assigned during pass 1, prints the error message \*SE, and continues.

At the completion of the pass 2 assembly process, the assembler halts with a one in the program counter (P register). The assembler is now ready to accept another assembly with the same I/O devices as previously specified. If it is desired to change the I/O devices, it is necessary to reload, starting with the I/O section as outlined previously.

If it is necessary to restart the assembler, press STEP, set all registers to zero, and press computer RESET and RUN. The assembler halts with a one in the program counter (P register) and is now ready to accept another assembly.

## DAS 8A Operations

Load the program into memory using the binary loader. Prepare the system input (SI), system output (SO), and list output (LO) units, and set the SENSE switches as follows. On pass 1, set SENSE switch 1 only. On pass 2, reset SENSE switch 1 and, for listing output, set SENSE switch 2, or, for binary object output, set SENSE switch 3. To begin assembly, RUN at address 0000001.

Pass 2 can be restarted or repeated without repeating pass 1.

END terminates both passes 1 and 2 by executing a HALT. After pass 1 is terminated, begin pass 2 by resetting the I/O devices, setting the SENSE switches as above, and pressing RUN. To obtain extra copies of the program, repeat pass 2 as desired. END is indicated by 017777 in the A register.

A MORE directive causes the computer to stop and wait until the input units are prepared and RUN is pressed. MORE is indicated by 0170017 in the A register.

Synchronization errors detected on pass 2 indicate that the value of a label on that pass does not agree with the value it had on pass 1. Such errors are due to source tape misreadings. They halt the assembly, as indicated by 0777 in the A register. To continue, press RUN. The assembler resets the location counter value to that assigned during pass 1, prints the error message \*SE, and continues.

binary output (BO), and list output (LO) units, and set the SENSE switches as follows. On pass 1, set SENSE switch 1 only. On pass 2, reset SENSE switch 1 and, for listing output, set SENSE switch 2, or, for binary object output, set SENSE switch 3. To begin assembly, RUN at address 0000001.

## DAS MR Operations

Since DAS MR is used within MOS and uses the MOS I/O control system, the I/O devices can be defined as required (see the Master Operating System Reference Manual).

DAS MR inputs the symbolic source statements from the processor input logical unit (PI) in alphanumeric mode, performs a pass 1 assembly, and outputs them in the same mode on the processor output logical unit (PO). When END is detected, pass 1 terminates, the assembler backs up the number of output symbolic source statements, and begins pass 2. This pass inputs the statements from the system scratch logical unit (SS), produces a listing on the LO unit, and a binary object program on the BO unit. Note that PO and SS must be the same magnetic tape, drum, or disc unit.

If a listing is not wanted, use the following directive to the MOS executive when requesting the assembly:

```
/ASSEMBLE      N
```

If a binary object program is not wanted, use the following directive:

```
/ASSEMBLE      B
```

If the memory map portion (symbol table, external names, and entry names) of the assembly listing is not wanted, use the following directive:

```
/ASSEMBLE      M
```

To read the same physical symbolic source statements for both assembly passes, use the following directives to the MOS executive:

```
/ASSIGN          PO = DUM,SS = PI
/ASSEMBLE
```

The PO symbolic source statement output serves as a copy of the statements, and can be the input for another assembly.

| <div>Memory Size</div> <div>Assembler Version</div> | 4K  | 8K   | > 8K            |
|-----------------------------------------------------|-----|------|-----------------|
| DAS 4A                                              | 150 | 1450 | 1450 + n (1300) |
| DAS 8A                                              | -   | 440  | 440 + n (800)   |
| DAS MR                                              | -   | 20   | 20 + n (800)    |

n = number of 4K memory increments above 8K

Figure 1-9. Symbol Table Capacities

# **Binary Load/Dump Program (BLD II) Reference Manual**







## SECTION I – BINARY LOAD/DUMP

The binary load/dump program allows the user to load object programs from a paper tape or TTY reader, or to punch the binary contents of memory on paper tape in a reloadable format.

### Location of BLD II

BLD II is loaded using the manual bootstrap routine or the automatic bootstrap loader (ABL) option. Once loaded into memory, BLD II relocates itself into the upper part of the highest 4K memory module unless the operator specifies a different 4K memory module.

Initially, BLD II occupies addresses 07000 to 07755. By residing in these locations, it does not interfere with the bootstrap loader occupying addresses 07756 to 07776. Immediately after loading, BLD II relocates to occupy addresses 0x7400 through 0x7755. x denotes the 4K memory module in which BLD II relocates as follows:

| X = | Memory Module |
|-----|---------------|
| 0   | 4K            |
| 1   | 8K            |
| 2   | 12K           |
| 3   | 16K           |
| 4   | 20K           |
| 5   | 24K           |
| 6   | 28K           |
| 7   | 32K           |

Entry to BLD II to read object tapes is always 0x7600

|   |     |     |     |     |     |
|---|-----|-----|-----|-----|-----|
| 0 | xxx | 111 | 110 | 000 | 000 |
|---|-----|-----|-----|-----|-----|

and entry to punch object tapes is 0x7404.

|   |     |     |     |     |     |
|---|-----|-----|-----|-----|-----|
| 0 | xxx | 111 | 100 | 000 | 100 |
|---|-----|-----|-----|-----|-----|

### Program Options Prior to Loading

The operator has four options prior to loading the binary load/dump program tape.

## binary load/dump

- a. No SENSE switches set: The program accepts input from the device specified in the entered bootstrap routine and stores the program in the highest 4K memory module. After reading the program in, the computer halts with the P register set to the entry address (0x7600) of the binary loader routine and the A, B, and X registers cleared.
- b. SENSE switch 1 set: This allows the operator to select any 4K memory module in which the program is to operate. After reading the program in, the computer halts with the P register set to 07013. The operator must enter in the A register the number (0 through 7) specifying the memory module in which the program is to reside. By pressing RUN, the operator initiates the relocation and the computer halts as in item (a) above.
- c. SENSE switch 2 set: This adjusts the dump routine for TTY punch output.
- d. SENSE switch 3 set: This allows the operator to splice an object program to the BLD II program tape, load the BLD II program and the object program and execute the object program without further intervention.

## Object Program Loading Options

The binary load/dump program establishes the input reader (TTY or high-speed paper tape) and the output punch devices by interrogating the bootstrap loader. For this reason, the binary load/dump program is adjusted to accept object program tapes from only the reader specified by the bootstrap routine.

However, setting SENSE switch 2 prior to loading the load/dump program adjusts the program for TTY punch regardless of the bootstrap-routine-specified I/O devices.

## Punching Tapes of Memory Contents

To punch a tape from memory to the high-speed paper tape punch, SENSE switch 2 must not be set when the load/dump program is entered. To punch a tape from memory to the TTY punch, SENSE switch 2 must be set when the load/dump program is entered if the input reader is a high-speed paper tape device. The operator can specify that tapes be punched in binary format to load using the binary loader or punch the binary loader in bootstrap-loadable format.

To punch a tape in binary format:

- a. Set the P register to 0x7404.
- b. Set the A register to the address of the first word to be punched.
- c. Set the B register to the address of the last word to be punched.
- d. Set the X register to the address of the first instruction to be executed at load time. (To punch noncontiguous areas, set the X register to -1(0177777) for all but the last area to be punched.)

To punch the BLD II program in a bootstrap-loadable format, set the P register to 0x7400. Set the A and B registers to zero and the X register to nonzero.

## Loading the Binary Load/Dump Program

- a. Enter the bootstrap loader and turn off repeat switch on 620/f.
- b. Clear the I register.
- c. Set the P register to 007770.
- d. Set the X register to 007000.
- e. Set SENSE switches, if required.
- f. Turn on the paper tape reader. (If 33/35 ASR is used, turn TTY to local and press CRTL D, T, and Q, then turn back on line.)
- g. Position the BLD II tape in reader with the first data frame after the eight-level punches under the high-speed reader head or under the read station of the TTY reader. (Place reader control lever of TTY to stop position.)
- h. Press RUN or set STEP/RUN to RUN and press START. (If 33/35 ASR, press RUN or START; then set the control lever to RUN position.) Load is complete when the computer changes to step mode.
- i. If SENSE switch 1 is set, reset SENSE switch 1 and clear the A register.
- j. Enter the appropriate octal value in the A register to relocate load/dump program, if applicable. Press RUN or START.
- k. The computer will halt in step mode with the P register containing 0x7600 unless SENSE switch 3 was set. With SENSE switch 3 set, the computer will execute the object program.
- l. Remove the BLD II program tape and reset SENSE switch 2, if applicable.

## Loading the Object Program Tape

Object program tapes can be loaded immediately after the binary load/dump program because the P register is set to the load starting address (0x7600). For all subsequent loadings, assure that P is set to 0x7600.

### Verification

To ensure that an object tape contains no errors before loading into core memory, the binary load/dump program has an option that performs only check-sum error checking. To use this option:

## binary load/dump

- a. Turn on the reader and position the object tape in the reader.
- b. Set P register to x7600.
- c. Enter 100000 in the A register.
- d. Clear the I register.
- e. Press RUN or set STEP/RUN to RUN and press START.
- f. A verify with no errors is indicated by the computer halting with:

|            |   |                   |
|------------|---|-------------------|
| P register | = | 0x7600            |
| A register | = | 100000            |
| B register | = | 000000            |
| X register | = | execution address |

- g. A verify with a check-sum error is indicated by the computer halting with:

|            |   |                                              |
|------------|---|----------------------------------------------|
| P register | = | 0x7600                                       |
| A register | = | 100000                                       |
| B register | = | 177777                                       |
| X register | = | load address of last correct<br>address read |

- h. To retry the check-sum error record, reposition the object program tape at the previous visual aid and press RUN. If a check-sum error is read again, check each character in the record. An error has been made in punching or the tape may be slightly torn.

## Load Program and Halt

To load the object program and halt:

- a. Turn on the reader and position the object tape in the reader.
- b. Clear the A, B, X, and I registers.
- c. Set the P register to 0x7600.
- d. Press RUN or set STEP/RUN to RUN and press START.
- e. Correct loading will be indicated by:

|            |   |                           |
|------------|---|---------------------------|
| P register | = | 0x7600                    |
| A register | = | 000000                    |
| B register | = | 000000                    |
| X register | = | program execution address |

- f. A check-sum error is indicated by the same conditions as above.

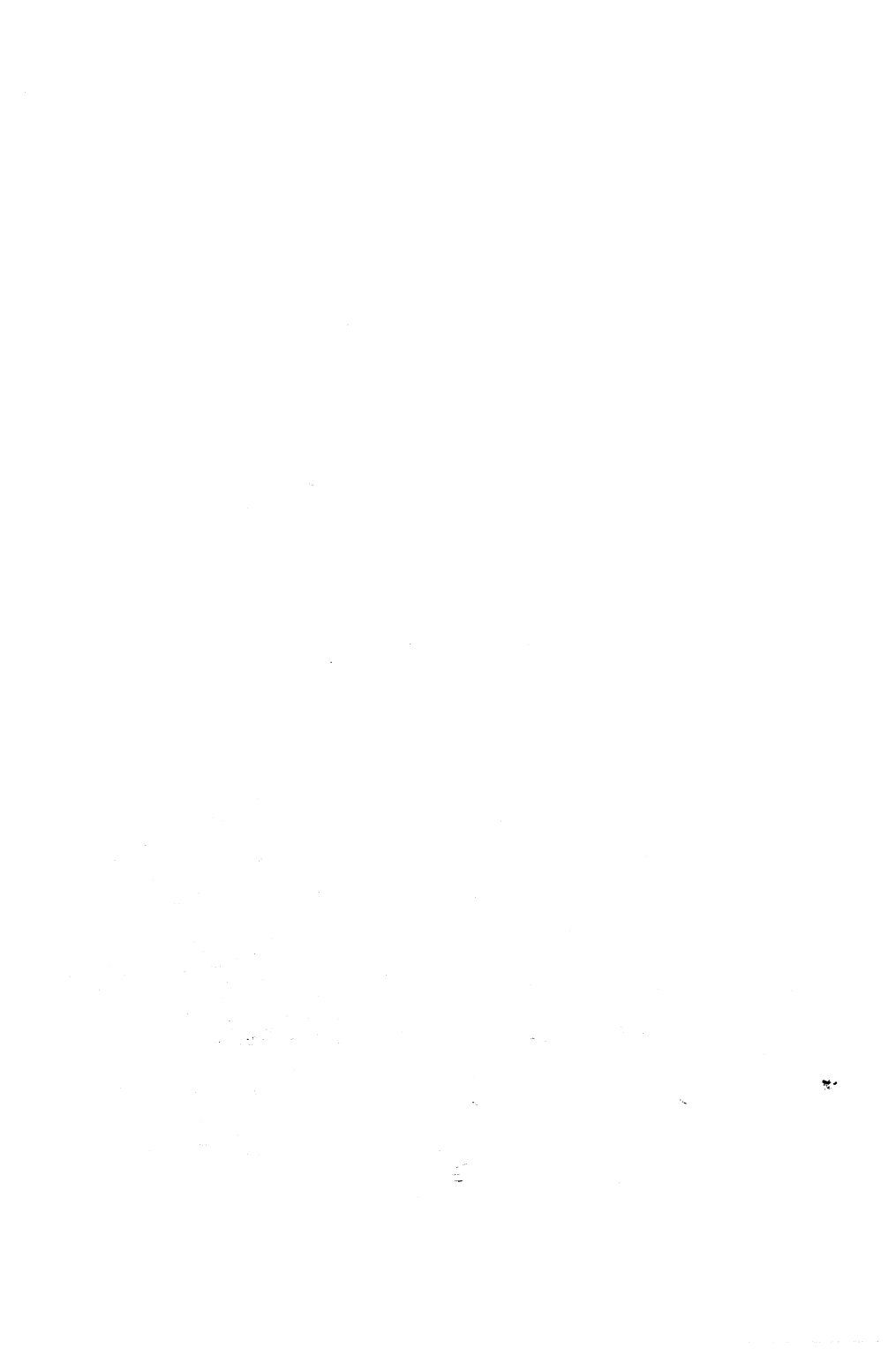
### **Load the Program and Execute**

Programs can be loaded and executed using the steps in the previous section except, in step b, set the A register to 000001 (any positive number).

### **Punching Program Tapes**

With the binary load/dump program loaded and paper tape punch on, areas of memory can be punched in object-tape-loadable format.

- a. Set the A register to the beginning address of area to be punched.
- b. Set the B register to the last address to be punched.
- c. Set the X register to the address of first instruction to be executed at load time, or if noncontiguous memory areas are to be punched, set the X register to - 1 (177777).
- d. Set the P register to 0x7404.
- e. Clear the I register.
- f. Press RUN or set STEP/RUN to RUN and press START.
- g. Tape will be punched and the computer halts with registers unaltered. If additional areas are to be punched, perform steps a through f above, entering the new areas in the A and B registers. Prior to punching the last area, set the X register to the address of the first instruction to be executed at load time.



# **Debugging Program (AID II) Reference Manual**







## SECTION I – DEBUGGING PROGRAM

Varian provides the AID II on-line debugging program for all Varian 620 systems. AID II is the most advanced of the debugging aid packages for the Varian 620 family of computers.

AID II provides the user with software to facilitate program checkout. By entering AID commands on the TTY, the operator can display and alter the contents of any memory address or block. Programs can be trapped into and out of user-selected blocks and can be searched for specific conditions. Any program can be entered, monitored, and altered from the TTY.

As an added feature, data can also be dumped onto magnetic tape, punched out on paper tape, or printed on the TTY keyboard. Object programs can thus be converted from one media to another, simply and directly.

### Loading AID II

AID II is loaded using the binary load/dump routine. Once loaded, AID II is positioned in the memory addresses just below the binary load/dump routine. It always occupies the highest memory addresses regardless of memory size. AID II occupies 06000-07377 memory addresses.

### Register and Memory Modification

With AID entered and the computer in the run mode, the following entries on the TTY keyboard produce the indicated results. Entering a RUBOUT character cancels entries, or if an operation is in progress, terminates it.

#### Operator Enters:

#### Operation and Results

- |       |                                                                                                                                                                                      |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| A.    | The program types the contents of the pseudo-A register on the TTY keyboard. If the contents are to be changed, type the desired number and a period; otherwise, type only a period. |
| B.    | Same as above for the pseudo-B register.                                                                                                                                             |
| X.    | Same as above for the pseudo-X register.                                                                                                                                             |
| Cxxx. | Same as above for memory contents xxx. By typing a comma (,) instead of a period, the operator requests the next memory address for display.                                         |
| Gxxx. | Loads the contents of the pseudoregisters into the respective A, B, and X registers and starts execution at address xxx.                                                             |

|           |                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Vxxx.     | The program starts typing memory address contents at xxx and continues until a RUBOUT character is entered. The left column is the base address in octal. The contents of eight memory addresses are typed across the page in ascending order. The first number in the next line indicates the base address for the next eight memory address contents.                                                                |
| Sx,y,z,m, | Search through memory starting at x and ending at y for the value of z masked by value of m. A masked-search searches for comparison with value of z for each bit corresponding to a one in the m value. Each time the values compare, the address and value are printed on the TTY. By typing an N instead of a mask value, the program searches for the negative value of z. Omission of m assumes an all-ones mask. |
| Ix,y,z.   | The program stores the value of z in all memory addresses starting at address x and ending at address y.                                                                                                                                                                                                                                                                                                               |
| Ty,x.     | Preset registers and go to location x. If and when location y is reached, save and type location y; the contents of y, and the current values of the registers. (Trap to y from x.)                                                                                                                                                                                                                                    |
| Ty,       | Continue trap from last breakpoint location to new breakpoint location y. (Present and save registers as before.)                                                                                                                                                                                                                                                                                                      |

**WARNING:** An incomplete trap (no return to AID II) will leave the object program with the instruction at the breakpoint location changed to a return jump to the AID II trap return. These locations should be restored to their original values before further use of the trap function to ensure proper results.

## Handling Paper Tape

The paper tape reader and punch can be controlled through AID II for some operations. With AID II entered and the computer in run mode, the paper tape system responds as indicated.

### Operator Enters:

### Operation and Results

|          |                                                                                                 |
|----------|-------------------------------------------------------------------------------------------------|
| Dx,y,z,. | Punch a program tape from the contents of address x to the address y, with execution address z. |
|          | X = -1 allows punching of noncontiguous areas.                                                  |

Lm,                      If the value m is one and no check-sum errors were encountered, the program is executed. If the value of m is zero and no check-sum errors were encountered, the TTY prints the contents of the A, B, and X registers, respectively, on the next line. The printout for the A and B registers will be zeros. The X register printout is the execution address of the object program. If a check-sum error is encountered, the B printout contains - 1, and the X printout contains the address of the last record from the tape. If M = - 1 the operation is the same as zero except the object tape is verified but not loaded into core.

## Magnetic Tape Handling

Data can be manipulated from and to magnetic tape through AID II. With AID II entered and the computer in run mode, the magnetic tape responds as indicated.

| Operator Enters: | Operation and Results                                                                                                                                                                                                                                                                                                      |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Eu.              | Write a file mark on magnetic tape unit u.                                                                                                                                                                                                                                                                                 |
| Fn,u,            | Skip to file n on magnetic tape unit u.                                                                                                                                                                                                                                                                                    |
| N,               | Skip to the next file on the magnetic tape unit previously designated.                                                                                                                                                                                                                                                     |
| Pu,              | Backspace one record on magnetic tape unit u.                                                                                                                                                                                                                                                                              |
| Ru, (or Ru.)     | Read an object magnetic tape into memory from magnetic tape unit u. A period causes a load and return to AID II. A comma causes a load and execute. If an uparrow is printed, a file mark was read on the tape. If an octal number is typed out, a parity error was encountered and the address of the error is indicated. |
| Wx,y,z,u         | Write an object magnetic tape from memory where x is the starting address, y is the ending address, z is the execution address, and u designates the magnetic tape unit.                                                                                                                                                   |



# **Source Program Editor (EDIT) Reference Manual**





## SECTION I — SOURCE PROGRAM EDITOR

EDIT is a standard program that permits the programmer to prepare and write symbolic programs and generate a symbolic program tape. EDIT is very flexible in that the symbolic program can be typed, on line, on the teletype keyboard and stored directly in the core memory. Then, using EDIT commands, the program can be listed (printed).

EDIT also adds, corrects, and deletes any portion of the symbolic program. When the program is correct and ready to be assembled or compiled, EDIT can generate a symbolic program tape of the stored program.

EDIT is on punched paper tape in binary format, and can be loaded into memory using BLD II (in the load-and-go mode).

### Using the notation

H = high speed paper tape system

T = teletype paper tape system

answer the following questions printed on the teletype.

SOURCE PAPER TAPE PROGRAM

INPUT DEVICE (H or T)

(Type an H or a T.)

OUTPUT DEVICE (H or T)

(Type an H or a T.)

EDIT will dynamically adapt to this configuration and enter the instruction mode. (A carriage return, line feed and asterisk will be typed. EDIT can be restarted at any time by setting the instruction and P registers to zero and pressing RUN. But it can only be reconfigured on loading.

Initially, EDIT is in the instruction mode, i.e., it can accept instructions. Anything typed by the user is interpreted as a command to EDIT, but EDIT accepts only legal instructions. EDIT ignores other inputs and types a question mark.

When not in the instruction mode, EDIT is in test mode, i.e., all characters input from the keyboard or tapes are interpreted as text to be put into the text buffer in the manner specified by a preceding EDIT instruction. Figure 1-1 illustrates the transfer of EDIT from one mode to the other.



## Search and Find

A very convenient feature available with EDIT is the SEARCH feature, which searches a line of text for a specified character. When you type a line number followed by S, EDIT waits for the input of the character for which it is to search. When EDIT locates the character and types it, typing stops. You can then input all or any combination of the following:

- a. New text (terminate the line with the return key).
- b. Rubout to delete the entire line to the left.
- c. Return to delete the entire line to the right.
- d. - to delete from right to left one character for each -  
- typed (A - is echoed for each - typed).
- e. CTRL/CC to search for the next occurrence of the search character.
- f. CTRL/bell to change the search character to the next character typed by the programmer.

Another feature is the FIND that searches for a string of characters in the text. When you type F (followed by return) and a string of characters, terminated by a return, the text is searched for that string. When it is found, the line in which it first appears is typed and the current line pointer is set to that line. The program returns to instruction mode. If the string did not appear, no action is taken and the program returns to instruction mode.

## Error Detection

EDIT checks all instructions for nonexistent information and incorrect formatting. When an error is detected, EDIT types a question mark and ignores the instruction.

## Special Keys and Instructions

EDIT is controlled by the use of special keys and instruction. Certain keys can be used in either instruction or text mode with the mode of operation determining the function of each key as shown in figure 1-2.

EDIT instructions are given in instruction mode. There are three basic types of instructions - input, editing, and output (figure 1-3). Instructions are executed when return is pressed.

All instructions are executed when the return key is depressed. Any other character causes the instruction to be ignored and a question mark typed.

| Instruction | Meaning                                                                                                               |
|-------------|-----------------------------------------------------------------------------------------------------------------------|
| A           | Add incoming text from the keyboard to the text buffer immediately following the text currently etched in the buffer. |
| R           | Read incoming text from the tape reader and append it to the text currently stored in the buffer.                     |
| L           | List entire text buffer. Specify one line or a group of lines, including line numbers.                                |
| C           | Change a line. Precede the instruction with the decimal line number or line numbers of the lines to be changed.       |
| I           | Insert into text buffer. Specify the line where the inserted text is to begin.                                        |
| D           | Delete from text buffer. Specify the line or group of lines to be deleted.                                            |
| P           | Punch text buffer. Specify one line, a group of lines, or the entire text buffer.                                     |

**Figure 1-1. Transfer between EDIT Modes**

| Instruction Mode                                                    | Text Mode                                                                    |
|---------------------------------------------------------------------|------------------------------------------------------------------------------|
| Execute preceding instruction                                       | Enter line in text buffer                                                    |
| Illegal                                                             | Cancel line to the left margin                                               |
| Cancel preceding instruction                                        | Delete to the left one character for each depression (A is echoed each time) |
| Respond with * and remain in instruction mode                       | Return to instruction mode and print                                         |
| Value equal to decimal                                              | Legal text character                                                         |
| Value of current line (used alone or with - and A number, e.g., - B |                                                                              |
| Value equal to number of last line in buffer (used as an argument)  | Legal text character                                                         |
| List next line                                                      |                                                                              |
| Used with or to detain their value                                  |                                                                              |
|                                                                     | Produces a tab interpreted as seven spaces or output                         |

Figure 1-2. EDIT Key Functions

| Instruction | Function                                                                                                                                                  |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Input       |                                                                                                                                                           |
| A           | Add incoming text from keyboard to text buffer.                                                                                                           |
| R           | Add incoming text from paper tape reader to text buffer                                                                                                   |
| Editing     |                                                                                                                                                           |
| L           | List entire text buffer                                                                                                                                   |
| NL          | List line N                                                                                                                                               |
| M,NL        | List lines M through N inclusive with line numbers                                                                                                        |
| NC          | Change line N                                                                                                                                             |
| M,NC        | Change lines M through N inclusive                                                                                                                        |
| I           | Insert before first line                                                                                                                                  |
| NI          | Insert before Line N                                                                                                                                      |
| K           | Delete entire text buffer                                                                                                                                 |
| ND          | Delete line N                                                                                                                                             |
| M,ND        | Delete lines M through N inclusive                                                                                                                        |
| G           | Print next line beginning with an alphabetic character<br>(if none, no action taken)                                                                      |
| NG          | Print next line after N beginning with an alphabetic<br>character (if none, no action taken)                                                              |
| S           | Search buffer for character specified after return.<br>Allow modification (search character is not echoed on<br>printer), and print lines while searching |

Figure 1-3. EDIT Instructions (1 of 2)

|            |                                                                                                                                                  |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| NS         | Search line N, as above                                                                                                                          |
| M,NS       | Search M through N inclusive, as above                                                                                                           |
| F<br>XXXX  | Search text for string XXXX (up to 72 characters),<br>and type out line in which it appears (if string<br>not found, return to instruction mode) |
| NF<br>XXXX | Search from line N as above                                                                                                                      |
| Output     |                                                                                                                                                  |
| P          | Punch entire text buffer                                                                                                                         |
| NP         | Punch line n                                                                                                                                     |
| M,NP       | Punch lines M through N inclusive                                                                                                                |
| T          | Punch about 20 inches of leader/trailer tape                                                                                                     |

Note: M and N are decimal integers, with  $M \leq N$ .

**Figure 1-3. EDIT Instructions (2 of 2)**

# Mathematical Package Reference Manual





## SECTION I – MATHEMATICAL PACKAGE

This section is intended to acquaint the programmer with the standard subroutine library. The library is detailed in Varian Subroutine Descriptions (98 A 9902 044).

If a subroutine requires only one parameter or argument, programmed entry is made by first loading the desired parameter into the A register, or A and B registers for a double or floating point entry, and then executing a jump and mark instruction to the subroutine.

Where more than two input parameters are required, the parameters are entered into the program following the jump to the subroutine. The sequence of instructions in Figure 1-1 is used.

### **Fixed-point single-precision multiply**

Identification: XMUL.

Provides the software version of the (optional) hardware multiply instruction. Uses recursive addition of multiplicand with shifting.

### **Fixed-point single-precision divide.**

Identification: XDIV.

Provides the software version of one (optional) hardware divide instruction. The true remainder and quotient are delivered to the A register and B register, respectively.

### **Fixed-point double-precision 2's complement**

Identification: XDCO.

Takes the 2's complement of the double-precision number in the A register and B register. The X register is unchanged. The argument is complemented and the low-order bits are tested for a carry condition.

### **Fixed-point double-precision add**

Identification: XDAD.

Adds a double-precision number whose high-order address is in the calling sequence to the double-precision numbers in the A register and B register. The X register is unchanged. Low-order words are added first and any carry generated is added to the high-order sum.



| Location  | Instruction   | Remarks                                           |
|-----------|---------------|---------------------------------------------------|
| P         | Jump and mark | Jump to subroutine.                               |
| P + 2     | Parameter     | Parameters or parameter locations for subroutine. |
| P + 3     | Parameter     | Parameters or parameter locations for subroutine. |
| P + 4     | Parameter     | Parameters or parameter locations for subroutine. |
| P + n     | Parameter     | Parameters or parameter locations for subroutine. |
| P + n + 1 | Normal return | Continuation of program.                          |

Figure 1-1. Sequence of Instructions for Entering Multiple Parameters

**Fixed-point double-precision subtract**

Identification: XDSU.

Subtracts a double-precision number whose high-order address is in the calling sequence from the double-precision number in the A register and the B register. The X register is unchanged.

**Fixed-point double-precision multiply**

Identification: XDMU.

Multiplies the double-precision number whose high-order address is in the calling sequence times the double-precision number in the A register and the B register. The X register is unchanged. Uses double-precision addition of partial products.

$$(A + a*2^{l-5}) (B + b*2^{l-5}) = AB*2^0 + Ab*2^{l-5} + aB*2^{l-5}$$

**Fixed-point double-precision divide**

Identification: XDDI.

Divides the double-precision number in the A register and B register by the double-precision number whose high-order address is in the calling sequence. The X register is unchanged.

**Absolute value, floating point (type real)**

Identification: ABS.

Takes the absolute value of the floating-point (real) quantity in the A, B registers, returning the result to the A, B registers. The absolute value of a is defined as  $-a$  if a is negative, as a if a is not negative.

**Absolute value, fixed-point (type integer)**

Identification: IABS.

Takes the absolute value of the 16-bit signed integer in the A register and returns the result to the A register. The absolute value of a is defined as  $-a$  if the a is negative and a if a is non-negative.

**Transfer of sign, fixed-point (type integer)**

Identification: ISIGN.

Applies the sign of the called (second) parameter to the quantity in the accumulator (first parameter). The parameters and result are fixed-point quantities. Uses \$SE.

## **mathematical package**

### **Copy sign**

Identification: SIGN.

Sets sign of floating-point number equal to that of argument. Output in A, B registers. Uses \$SE.

### **Separate mantissa**

Identification: \$FMS, \$FSM.

Separates a positive floating-point number into characteristic and mantissa. Output in A, B (mantissa) and X (characteristic) registers.

### **Floating point number to integer**

Identification: \$HS.

Converts a floating point number to an integer. Uses \$SE.

### **Normalize**

Identification: \$NML.

Normalizes a double-precision number. Shifts to sign and tests for sign set. Uses XDCO. Output in A, B registers. Flag for sign in X register.

### **Floating add**

Identification: \$QK.

Algebraically adds two floating-point numbers. \$QK and \$QL use common logic.

### **Floating subtract**

Identification: \$QL.

Computes difference of two floating-point numbers.

### **Floating-point multiply or divide**

Identification: \$QM, \$QN.

Multiplies two floating-point numbers. Divides one number by another. Separates the mantissa and use XDMU for multiply or XDDI for divide. Uses \$FMS, \$SE.

### **Integer number to floating-point number**

Identification: \$QS.

Floats an integer. Formats the absolute number to floating point and adjusts sign according to input. Uses \$SE.

### Fixed single-precision logarithm

Identification: XLOG.

Computes the natural logarithm of  $1 + X$ , where the single-precision quantity  $X$  is in the A register. If

$$0 \leq X < 1,$$

the result is returned to the A register; otherwise an error exit is taken without further action. Input and output are scaled by  $2^0$ . Uses a Chebychev polynomial of the fifth degree.

### Fixed single-precision exponential, positive arguments

Identification: XEXP.

Computes the exponential of  $X$ , located in the A register:

$$e^X, 0 \leq X < 1$$

$e^X$  is scaled  $2^2$ . The result is placed in the A register. The exponential is performed by means of a Chebychev polynomial of the fifth degree.

### Fixed single exponential, negative argument

Identification: XEXN.

Computes the exponential of  $X$ , located in the A register:

$$e^X, -1 \leq X < 0$$

$e^X$  is scaled  $2^0$ . The result is placed in the A register. (Also see purpose in subroutine XEXP.) The exponential was split into two subroutines, XEXP and XEXN, to increase scaling flexibility. The exponential is performed by means of a Chebychev polynomial of the fifth degree.

### Fixed single-precision square root (short)

Identification: XSQT.

Takes the unrounded square root of the quantity in the A register if it is non-negative. The result is returned to the A register. The A register is unchanged if the input is negative.

### Fixed single-precision sine

Identification: XSIN.

Takes the sine of the quantity  $X$  in the A register for range  $-\pi \leq X \leq \pi$ . The input is scaled by  $2^2$ . The output is returned to the A register, scaled by  $2^1$ . The output is returned to the A register, scaled by  $2^1$ .  $X$  is in radians. Uses a change of variable to  $y$  to reduce range from  $(-\pi, \pi)$  to  $(-\pi/2, \pi/2)$ . The change of variable is  $\sin x = \sin y$ .

$$y = |X - \pi/2| - \pi/2 \text{ if } X \geq 0$$

$$y = |X - \pi/2| + \pi/2 \text{ if } X < 0$$

The Taylor sine series, truncated to five terms, is used for sine  $y$ .

### Fixed single-precision cosine

Identification: XCOS.

Takes the cosine of the quantity  $X$  in A register from range  $-\pi \leq X \leq \pi$ . The input is scaled by  $2^2$  and the output is scaled by  $2^1$ . The output is returned to the A register. Uses a change of variable to  $y$  in order to reduce the range of the variable from  $(-\pi, +\pi)$  to  $-\pi/2, +\pi/2$ . Then  $\cos x = \sin y$ , where  $y = \pi/2 - X$ . The Taylor sine series, truncated to five terms, is used for  $\sin y$ .

### Fixed single-precision arctangent

Identification: XATN.

Takes the arctangent of the quantity  $X$  in the A register, where  $-1 < X < 1$ . The input is scaled times  $2^0$  and the output is scaled times  $2^0$ . Uses a Chebychev polynomial of seven terms. This polynomial is adequate for an 18-bit configuration.

### Single-precision polynomial

Identification: POLY.

A resident utility routine intended primarily to support the fixed-point single-precision mathematical subroutines requiring the evaluation of a polynomial in one variable of any finite degree. The polynomial is evaluated in Horner form.

### Natural log of floating-point number

Identification: ALOG.

Computes natural log of a floating-point number. Uses \$ER, \$QS, \$QK, \$QM, XDMU, XDAD, \$FMS, \$NML, XDDI, XDSU, \$SE routines.

### Arctangent of a floating-point number

Identification: ATAN.

Computes arctangent of radians in floating point. Uses \$QM, \$QL, \$QN, \$QK, \$SE routines.

### Cosine

Identification: COS.

Computes cosine of angle in floating-point radians. Computes sine of ( $\pi/2 - A$ ). Uses SIN, \$QL, \$SE. Output in A, B registers.

### Exponential

Identification: EXP.

Computes  $e^{**}A$ . A is floating-point number. Chebychev approximation uses XDMU, \$QK, \$QL, \$QM, \$QN, \$SE.

### Sine

Identification: SIN.

Computes sine of radians in floating point. First five terms of Taylor series expansion output in A, B registers. Uses \$NML, \$QM, XDMU, XDAC, \$SE, \$FMS.

### Square root

Identification: SQRT.

Computes square root of a floating-point number. Newton iteration three times. Uses \$SE, XDDI, \$FMS.

### Exponentiation of two integers

Identification: \$HE.

Computes  $I^{**}J$ , I and J are integers. Floats I and uses \$PE. Uses \$SE, \$QS, \$HS, \$PE.

### Exponentiation

Identification: \$PE.

Computes  $A^{**}I$ , A is a floating-point number, I is an integer. Uses \$QS, \$QE, and \$SE. Floats I and goes to  $A^{**}B$  (\$QE).

### Exponentiation

Identification: \$QE.

Computes  $A^{**}B$ , antilog of  $B \log A = e^{**} (B \log A)$ . Uses ALOG, EXP, \$SE.

### **Fixed single-precision integer binary-to-decimal conversion**

Identification: XBTD.

Converts the absolute value of the integer in the A register, modulo 10,000, to a four digit decimal coded integer in the B register. The input is retained in the A register and the X register is unchanged. The output range is 0 through 9999 inclusive. Uses successive division of binary integer by 10 with concatenation of remainders.

### **Fixed single-precision integer decimal-to-binary conversion**

Identification: XDTB.

Converts the four-digit binary-coded decimal integer in the A register to a binary integer in the B register. The input is retained in the A register with the X register unchanged. The input range is +0 through +9999 inclusive. Uses successive multiplication of digits by powers of 10 with accumulation.

$$B = ((10D_3 + D_2)_{10} \rightarrow 1) 10 + D_0.$$

### **Converts EBCDIC to Hollerith**

Identification: SA01.

Converts an EBCDIC character in bits 0 through 7 of the A register to 029 Hollerith code in bits 0 through 11 of the A register.

### **Convert Hollerith to EBCDIC**

Identification: SB01

Converts an 029 Hollerith character in bits 0 through 11 of the A register to its corresponding EBCDIC code in bits 0 through 7 of the A register.

### **Converts EBCDIC to ASCII+**

Identification: SC01

Converts an 8-bit EBCDIC character in bits 0 through 7 of the A register to its equivalent 8-bit ASCII code in bits 0 through 7 in the A register.

# **FORTRAN IV Reference Manual**







## **TABLE OF CONTENTS**

### **SECTION 1 INTRODUCTION**

|                     |     |
|---------------------|-----|
| General.....        | 1-1 |
| Character Set ..... | 1-2 |
| Line Format .....   | 1-3 |

### **SECTION 2 DATA**

|                  |     |
|------------------|-----|
| General.....     | 2-1 |
| Data Types ..... | 2-1 |
| Data Names ..... | 2-1 |
| Variables .....  | 2-2 |
| Constants.....   | 2-2 |
| Arrays.....      | 2-7 |

### **SECTION 3 SPECIFICATIONS AND STATEMENTS**

|                            |     |
|----------------------------|-----|
| General.....               | 3-1 |
| Dimension Statement .....  | 3-1 |
| Common Statement .....     | 3-2 |
| Equivalence Statement..... | 3-5 |
| Type Statement .....       | 3-6 |

### **SECTION 4 EXPRESSIONS AND ASSIGNMENTS**

|                                       |     |
|---------------------------------------|-----|
| Arithmetic Expressions.....           | 4-1 |
| Arithmetic Assignment Statement ..... | 4-4 |
| Logical Assignment Statement.....     | 4-7 |
| Logical Expressions.....              | 4-8 |

## SECTION 5 CONTROL STATEMENTS

|                              |     |
|------------------------------|-----|
| General.....                 | 5-1 |
| Go To Statements.....        | 5-1 |
| Arithmetic If Statement..... | 5-4 |
| Call Statement.....          | 5-6 |
| Return Statement.....        | 5-7 |
| Continue Statement.....      | 5-7 |
| Pause Statement.....         | 5-8 |
| Stop Statement.....          | 5-8 |
| Do Statement.....            | 5-9 |

## SECTION 6 INPUT/OUTPUT STATEMENTS

|                                          |      |
|------------------------------------------|------|
| General.....                             | 6-1  |
| Input/Output Lists.....                  | 6-1  |
| Simple Lists.....                        | 6-2  |
| Do-Implied Lists.....                    | 6-2  |
| Read Statements.....                     | 6-3  |
| Write Statements.....                    | 6-4  |
| Rewind Statement.....                    | 6-5  |
| Backspace Statement.....                 | 6-5  |
| Endfile Statement.....                   | 6-5  |
| Format Statement.....                    | 6-5  |
| Field Specifications.....                | 6-6  |
| F Conversion.....                        | 6-7  |
| E Conversion.....                        | 6-8  |
| D Conversion.....                        | 6-9  |
| I Conversion.....                        | 6-9  |
| A Conversion.....                        | 6-10 |
| H Conversion.....                        | 6-11 |
| X Specification.....                     | 6-12 |
| L Format Code.....                       | 6-12 |
| G Format Code.....                       | 6-13 |
| Scale Factor P.....                      | 6-15 |
| / Specification.....                     | 6-18 |
| Repeat Specifications.....               | 6-19 |
| Format Control and Line Interaction..... | 6-21 |

## SECTION 7 PROGRAMS AND SUBPROGRAMS

|                               |      |
|-------------------------------|------|
| General.....                  | 7-1  |
| Main Programs.....            | 7-1  |
| Subprograms.....              | 7-1  |
| Statement Functions.....      | 7-8  |
| Intrinsic Functions.....      | 7-9  |
| Basic External Functions..... | 7-9  |
| Dummy Arguments.....          | 7-9  |
| Adjustable Dimensions.....    | 7-10 |
| External Statements.....      | 7-12 |

## SECTION 8 OPERATING INSTRUCTIONS

|                                         |      |
|-----------------------------------------|------|
| General.....                            | 8-1  |
| Compiler Operating Instructions.....    | 8-1  |
| Loader Operating Instructions.....      | 8-2  |
| Input/Output Device Specifications..... | 8-6  |
| Compiler Input Records.....             | 8-8  |
| Compiler Output Records.....            | 8-8  |
| Notification Errors.....                | 8-9  |
| Termination Errors.....                 | 8-10 |
| Optional Listings.....                  | 8-10 |
| Maps.....                               | 8-11 |
| I/O Device Selection.....               | 8-14 |

## LIST OF FIGURES

|            |                                              |      |
|------------|----------------------------------------------|------|
| Figure 1-1 | Sample FORTRAN Coding Forms.....             | 1-5  |
| Figure 7-1 | Intrinsic Functions.....                     | 7-12 |
| Figure 7-2 | Basic External Functions.....                | 7-14 |
| Figure 8-1 | Run-Time Map for Stand-Alone FORTRAN IV..... | 8-10 |



## SECTION 1 – INTRODUCTION

Varian **FORTRAN IV** is a programming system for the 620 family of computers and is comprised of a language, a library of subprograms, a compiler, and a run-time package (program). It is available in both stand-alone and operating system (MOS) configurations.

### GENERAL

The **FORTRAN IV** language is especially useful in writing programs for scientific and engineering applications that involve mathematical computations. In fact, the name of the language **FORTRAN** is derived from its primary use: **FOR**mula **TRAN**slating. Source programs written in the **FORTRAN** language consist of a set of statements constructed from the elements described in this publication. The **FORTRAN** compiler analyzes the source program statements and transforms them into an object program that is suitable for execution on the 620. In addition, when the **FORTRAN** compiler detects errors in the source program, appropriate error messages are produced. The Varian **FORTRAN IV** language is compatible with and encompasses the American National Standards Institute (ANSI) **FORTRAN** including its mathematical subroutine provisions. Any valid programs compiled and executed using basic **FORTRAN** subset may also be compiled and executed by the **FORTRAN IV** compiler. Equivalent results are ensured by:

- a. Common data formats.
- b. Common format code routines.
- c. Common mathematical subroutines.
- d. Common libraries.

The following **FORTRAN IV** features facilitate the writing of source programs:

- a. Scale Factor: The scale factor allows modification of data during conversion between internal and external representation.
- b. Variable Attribute Control: The attributes of variables and arrays can be explicitly specified by statements that
  - (1). Specify the number of words assigned to an item.
  - (2). Explicitly type a variable as integer, real, double precision, complex, or logical.
  - (3). Specify the dimension of arrays.
  - (4). Specify data initialization values for variables.

## character set

- c. Adjustable Array Dimensions: The dimensions of an array in a subprograms can be specified as variables. When the subprogram is called, the absolute array dimensions are substituted.
- d. Three Dimension Arrays: An array has one, two, or three dimensions.
- e. Six Character Variable Names: The name of a variable contains up to six characters.
- f. Function subprograms return results via the argument list.

## CHARACTER SET

A **FORTRAN** program unit is written using the following letters, digits, and special characters:

Letters:      A B C D E F G H I J K L M N O P Q R S T U V W X Y Z \$

Digits:        0 1 2 3 4 5 6 7 8 9

Special Characters:

|   |                   |
|---|-------------------|
|   | blank or space    |
| = | equals            |
| + | plus              |
| - | minus             |
| * | asterisk          |
| / | slash             |
| ( | left parenthesis  |
| ) | right parenthesis |
| , | comma             |
| . | decimal point     |

With the exception of the specific uses indicated in the following sections of this manual, a blank character has no meaning, and can be used freely by the programmer to improve the readability of the **FORTRAN** program.

The following special characters are classified as arithmetic operators and are significant in the unambiguous statement of arithmetic expressions:

|    |                               |
|----|-------------------------------|
| +  | addition or positive value    |
| -  | subtraction or negative value |
| *  | multiplication                |
| /  | division                      |
| ** | exponentiation                |

The special characters equals (=), open parenthesis ( ( ), close parenthesis ( ) ), comma (,), and decimal point (.), have specific application in the syntactical expression of the **FORTRAN** statement. The following sections of this manual qualify their use in particular statements and expressions.

In addition to the **FORTRAN** character set, the Varian **FORTRAN IV** system accepts the following characters in Hollerith fields:

|   |                |
|---|----------------|
| " | quotation mark |
| ↑ | uparrow        |
| ! | exclamation    |
| # | number sign    |
| % | percent        |
| & | ampersand      |
| ' | apostrophe     |
| : | colon          |
| ; | semicolon      |

## LINE FORMAT

A **FORTRAN** program consists of a series of statements divided into physical sections called lines that must be coded to a precise grammatical format. **FORTRAN** statements fall into two broad classes, executable and nonexecutable. Executable statements specify program action; nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement. The statements of a **FORTRAN** source program are normally written on a standard **FORTRAN** coding form.

Figure 1-1 is a sample **FORTRAN** coding form. The coding form includes 80 columns of information. Columns 73 through 80 are reserved for sequencing information, and have no effect upon the generated execution program. Columns 1 through 72 contain line information in the format described below.



Initial Line

The first line of each statement is called an initial line. A statement line consists of three fields: statement number field, continuation flag, and statement field. A statement can include an initial line and continuation lines. Statements can have up to 19 continuation lines as required subject to the following restrictions: DO statements must be wholly contained on an initial line; and the equals character (=) of a replacement statement or a statement function definition must appear on the initial line. An initial line can contain a statement label in columns 1 through 5. In this case, column 6 must contain a zero digit, blank, or space character; and columns 7 through 72 may contain all or part of a statement except for the restrictions noted.

Example

|   |   |   |                |    |    |    |    |    |    |
|---|---|---|----------------|----|----|----|----|----|----|
| 1 | 5 | 6 | 7              | 10 | 15 | 20 | 25 | 30 | 35 |
|   |   |   | A = .5 * C * D |    |    |    |    |    |    |

Statement Number

Numbers permit statements to be referenced by other portions of a program. A statement number is an integer value in the range 1 to 99999 (leading zeros or blanks are not significant). The initial line of each statement may be given a unique number in columns 1 through 5. The same number cannot be given to more than one statement in a program unit.

Example

|     |   |   |                |    |    |    |    |    |    |
|-----|---|---|----------------|----|----|----|----|----|----|
| 1   | * | 6 | 7              | 10 | 15 | 20 | 25 | 30 | 35 |
| 50  |   |   | A = .5 * C + D |    |    |    |    |    |    |
| 60  |   |   | A = .5 * C + D |    |    |    |    |    |    |
| 879 |   |   | A = .5 * C + D |    |    |    |    |    |    |

| PROGRAM           |   | MATRIX MULTIPLICATION |    | PAGE NO.       | OF |
|-------------------|---|-----------------------|----|----------------|----|
| PROGRAMMER        |   |                       |    | IDENTIFICATION |    |
| DATE              |   |                       |    | DATE           |    |
| C FOR COMMENT     |   |                       |    |                |    |
| Statement Label   | 1 | 5                     | 10 | 15             | 20 |
| FORTRAN STATEMENT |   |                       |    |                |    |
| 1                 |   |                       |    |                |    |
| 2                 |   |                       |    |                |    |
| 3                 |   |                       |    |                |    |
| 4                 |   |                       |    |                |    |
| 5                 |   |                       |    |                |    |
| 6                 |   |                       |    |                |    |
| 7                 |   |                       |    |                |    |
| 8                 |   |                       |    |                |    |
| 9                 |   |                       |    |                |    |
| 10                |   |                       |    |                |    |
| 11                |   |                       |    |                |    |
| 12                |   |                       |    |                |    |
| 13                |   |                       |    |                |    |
| 14                |   |                       |    |                |    |
| 15                |   |                       |    |                |    |
| 16                |   |                       |    |                |    |
| 17                |   |                       |    |                |    |
| 18                |   |                       |    |                |    |
| 19                |   |                       |    |                |    |
| 20                |   |                       |    |                |    |
| 21                |   |                       |    |                |    |
| 22                |   |                       |    |                |    |
| 23                |   |                       |    |                |    |
| 24                |   |                       |    |                |    |
| 25                |   |                       |    |                |    |
| 26                |   |                       |    |                |    |
| 27                |   |                       |    |                |    |
| 28                |   |                       |    |                |    |
| 29                |   |                       |    |                |    |
| 30                |   |                       |    |                |    |
| 31                |   |                       |    |                |    |
| 32                |   |                       |    |                |    |
| 33                |   |                       |    |                |    |
| 34                |   |                       |    |                |    |
| 35                |   |                       |    |                |    |
| 36                |   |                       |    |                |    |
| 37                |   |                       |    |                |    |
| 38                |   |                       |    |                |    |
| 39                |   |                       |    |                |    |
| 40                |   |                       |    |                |    |
| 41                |   |                       |    |                |    |
| 42                |   |                       |    |                |    |
| 43                |   |                       |    |                |    |
| 44                |   |                       |    |                |    |
| 45                |   |                       |    |                |    |
| 46                |   |                       |    |                |    |
| 47                |   |                       |    |                |    |
| 48                |   |                       |    |                |    |
| 49                |   |                       |    |                |    |
| 50                |   |                       |    |                |    |
| 51                |   |                       |    |                |    |
| 52                |   |                       |    |                |    |
| 53                |   |                       |    |                |    |
| 54                |   |                       |    |                |    |
| 55                |   |                       |    |                |    |
| 56                |   |                       |    |                |    |
| 57                |   |                       |    |                |    |
| 58                |   |                       |    |                |    |
| 59                |   |                       |    |                |    |
| 60                |   |                       |    |                |    |
| 61                |   |                       |    |                |    |
| 62                |   |                       |    |                |    |
| 63                |   |                       |    |                |    |
| 64                |   |                       |    |                |    |
| 65                |   |                       |    |                |    |
| 66                |   |                       |    |                |    |
| 67                |   |                       |    |                |    |
| 68                |   |                       |    |                |    |
| 69                |   |                       |    |                |    |
| 70                |   |                       |    |                |    |
| 71                |   |                       |    |                |    |
| 72                |   |                       |    |                |    |

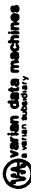


Figure 1-1. Sample FORTRAN Coding Form

**line format**

**Continuation Line**

Continuation lines are used when additional lines of coding are required to complete a statement originating on an initial line. There can be up to 19 continuation lines per statement with the exceptions previously noted for initial lines. In a continuation line, columns 1 through 5 are blank. Column 6 contains any character other than a zero, blank, or space. The continuation of the statement is in columns 7 through 72. Continuation lines follow an initial line or another continuation line only.

### Example

[illegible]

**Comment Line**

Any line with the character C in column 1 is identified as a comment line. Comments can appear anywhere in a program. All comment lines are ignored by a **FORTRAN** compiler, except for display purposes. Comments are in columns 2 through 72.

### Example

|                           |   |   |   |    |    |    |    |    |    |
|---------------------------|---|---|---|----|----|----|----|----|----|
| 1                         | 5 | 6 | 7 | 10 | 15 | 20 | 25 | 30 | 35 |
| C THIS IS A COMMENTS LINE |   |   |   |    |    |    |    |    |    |

## End Line

Any line containing the character blank in columns 1 through 6 and having only the character string **END** in columns 7 through 72, preceded by, interspersed with, or followed by blank characters, is recognized by the processor as an end line to inform the processor that it has reached the physical end of the program.

### Example

|   |   |   |     |    |    |    |    |    |    |
|---|---|---|-----|----|----|----|----|----|----|
| 1 | 5 | 6 | 7   | 10 | 15 | 20 | 25 | 30 | 35 |
|   |   |   | END |    |    |    |    |    |    |

## SECTION 2 – DATA

### GENERAL

Quantities exist as constants or variables. These are distinguished in **FORTRAN** to identify the nature and characteristics of the values encountered in program execution. A constant is a quantity whose value is explicitly stated. A variable is a numeric quantity referenced by name, rather than by its explicit appearance in a program statement. During the execution of a program, a variable can assume many different values.

### DATA TYPES

The Varian **FORTRAN IV** compiler recognizes the following types of data: integer, real double-precision, complex, logical, and Hollerith. Integer data are precise representations of integral values. Real data are approximations of real numbers. Both integer and real data may assume positive, negative, or zero values as follows (zero is considered neither positive nor negative):

|         | Range in<br>16-Bit Computers | Range in<br>18-Bit Computers |
|---------|------------------------------|------------------------------|
| Integer | $\pm 32,767$                 | $\pm 131,071$                |
| Real    | Approx. $10^{3.8}$           | Approx. $10^{3.8}$           |

### DATA NAMES

**FORTRAN** data (constants, variables, arrays, and array elements) are identified by names made up of letter or digit strings of one to six characters, the first character of which is a letter. (The character \$ is processed exactly like a letter, but it is reserved for Varian system names. To avoid conflict, therefore, it is advisable not to use the \$ character in names.) Data so identified are implicitly specified as being of type integer or real by the first character, although this can be changed by an explicit specification using a **TYPE** statement. Names beginning with the letters I, J, K, L, M, and N denote integers and other names denote real values.

Examples of integer names are:

I      I2A      MZXF      N5

Examples of real-number names are:

A      B2      F5M79      AAA

data

## VARIABLES

Variables are data whose values are derived and defined during program execution. They are identified by symbolic names of the appropriate type.

## CONSTANTS

Constant data are identified explicitly by giving their actual values. Constants do not change in value during program execution. They are specified as integer, real, double precision, complex, Hollerith, or logical constants.

### Integer Constants

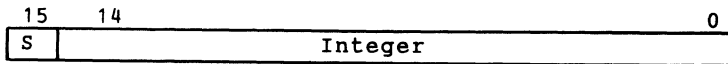
An integer constant is from one to five decimal digits without a decimal point. It can be preceded by a plus ( + ) or minus ( - ) sign.

#### Examples

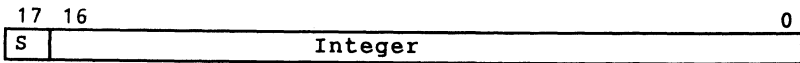
-217      -32767      +00327      512

In memory, an integer is stored in the format:

#### 16-Bit Computers



#### 18-Bit Computers



### Real Constants

A real single-precision constant has one to seven significant digits in any one of the following forms:

$\pm i.$        $\pm i.f$        $\pm .fE\pm e$   
 $\pm iE\pm e$        $\pm .f$        $\pm i.E\pm e$   
 $\pm i.fE\pm e$

where i, f, and e are each a string of decimal digits representing an integer, fraction and exponent, respectively. The sign character is optional. The decimal point (.) and E characters are obligatory in forms that specify them. If r represents any of the forms preceding  $E \pm e$ , i.e.,  $rE \pm e$ , then the real constant is interpreted as  $(r \times 10 \text{ to the power } \pm e)$ .

#### Examples

|         |            |       |
|---------|------------|-------|
| 17.     | -25.620E-1 | 0.0   |
| 51E1    | + .42      | -.479 |
| -479E-3 | .35E02     |       |

In memory, a real number is stored in the format:

#### 16-Bit Computers

|    |                |  |  |  |  |   |   |  |  |  |                 |  |  |  |   |
|----|----------------|--|--|--|--|---|---|--|--|--|-----------------|--|--|--|---|
| 15 | 14             |  |  |  |  | 7 | 6 |  |  |  |                 |  |  |  | 0 |
| S  | Characteristic |  |  |  |  |   |   |  |  |  | Mantissa (high) |  |  |  |   |
| 0  | Mantissa (low) |  |  |  |  |   |   |  |  |  |                 |  |  |  |   |

#### 18-Bit Computers

|    |                |  |  |  |  |  |  |   |   |  |                 |  |  |  |  |  |  |  |   |
|----|----------------|--|--|--|--|--|--|---|---|--|-----------------|--|--|--|--|--|--|--|---|
| 17 | 16             |  |  |  |  |  |  | 9 | 8 |  |                 |  |  |  |  |  |  |  | 0 |
| S  | Characteristic |  |  |  |  |  |  |   |   |  | Mantissa (high) |  |  |  |  |  |  |  |   |
| 0  | Mantissa (low) |  |  |  |  |  |  |   |   |  |                 |  |  |  |  |  |  |  |   |

If a real constant is specified with more significant digits than the precision real data allow, truncation occurs, and only the most significant digits within the range will be represented. For a 16-bit computer, 6+ significant decimal digits are represented for a single-precision real constant and 13+ significant decimal digits for a double-precision real constant. For an 18-bit computer, the corresponding precisions are 7+ and 15+ significant decimal digits.

#### Double-Precision Constants

A double-precision constant in a source statement is specified exactly as an E-format real constant, except that the letter E is replaced by D.

#### Examples

|             |       |          |
|-------------|-------|----------|
| -.3476.2D-4 | 28.D0 | 578D + 3 |
|-------------|-------|----------|

data

In memory, a double-precision number is stored in the format:

16-Bit Computers

|    |                 |   |   |                |
|----|-----------------|---|---|----------------|
| 15 | 14              | 8 | 7 | 0              |
| 0  | Zeros           |   |   | Characteristic |
| S  | Mantissa (high) |   |   |                |
| 0  | Mantissa (mid)  |   |   |                |
| 0  | Mantissa (low)  |   |   |                |

18-Bit Computers

|    |                 |   |   |                |
|----|-----------------|---|---|----------------|
| 17 | 16              | 8 | 7 | 0              |
| 0  | Zeros           |   |   | Characteristic |
| S  | Mantissa (high) |   |   |                |
| 0  | Mantissa (mid)  |   |   |                |
| 0  | Mantissa (low)  |   |   |                |

The exponent is eight bits long with a bias of 0200. If the mantissa is negative, the high-order word of the mantissa is in one's complement form. All four words are zero for the number zero.

Complex Constants

A complex constant is formed by an ordered pair of signed or unsigned real single-precision constants separated by a comma and enclosed in parentheses.

The real constants in a complex constant can be positive, zero, or negative (if unsigned, they are assumed to be positive). The first real constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number.

Examples

Valid Complex Constants

|                        |                              |
|------------------------|------------------------------|
| (-5.0E + 03.,16E + 02) | has the value -5000. + 16.0i |
| (4.0E + 03.,16E + 02)  | has the value 4000. + 16.0i  |
| (4.0E + 03.,16E + 02)  | has the value 4000. + 16.0i  |
| (2.1,0.0)              | has the value 2.1 + 0.0i     |

where i equals the square root of -1.

Invalid Complex Constants

|                 |                                                    |
|-----------------|----------------------------------------------------|
| (292704.1.697)  | the real part does not contain a decimal point     |
| (1.2E113.279.3) | the real part contains an invalid decimal exponent |
| (.003D4..005D6) | double-precision constants are invalid             |

In memory, a complex number is stored in the format:

16-Bit Computers

|                |    |                |  |   |                 |
|----------------|----|----------------|--|---|-----------------|
|                | 15 | 14             |  | 7 | 6               |
| Real Part      | S  | Characteristic |  |   | Mantissa (high) |
|                | 0  | Mantissa (low) |  |   |                 |
| Imaginary Part | S  | Characteristic |  |   | Mantissa (high) |
|                | 0  | Mantissa (low) |  |   |                 |

18-Bit Computers

|                |    |                |  |   |                 |
|----------------|----|----------------|--|---|-----------------|
|                | 17 | 16             |  | 9 | 8               |
| Real Part      | S  | Characteristic |  |   | Mantissa (high) |
|                | 0  | Mantissa (low) |  |   |                 |
| Imaginary Part | S  | Characteristic |  |   | Mantissa (high) |
|                | 0  | Mantissa (low) |  |   |                 |

Hollerith Constants

A Hollerith constant has the form: nHccc... where n is the number of characters in the constant and c is a character in the **FORTRAN** character set (page 1-2). It can be used only in the data initialization statement and in the argument list of a **CALL** statement.

Examples

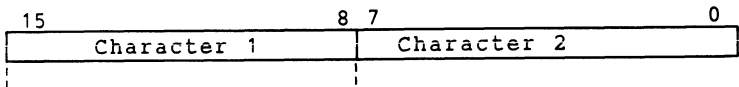
4HABCD  
9HTEST CASE



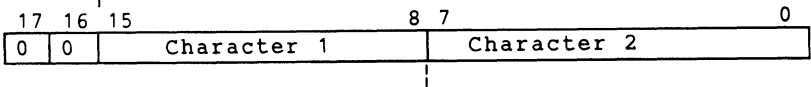
data

In memory, a Hollerith constant is stored in the format:

16-Bit Computers



18-Bit Computers



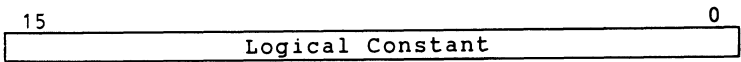
For Hollerith constants containing an odd number of characters, the last word contains the last character of the constant as character 1 and a blank as character 2.

Logical Constants

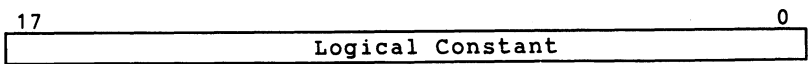
A logical constant specifies the logical value of a variable. There are two logical values: `.TRUE.` and `.FALSE.`. Each must be preceded and followed by a period as shown. The logical constants `.TRUE.` and `.FALSE.` specify that the value of the logical variable with which they are associated is true or false, respectively (page 4-4).

In memory, a logical constant is stored in the format:

16-Bit Computers



18-Bit Computers



where `.TRUE.` is stored as zero and `.FALSE.` is stored as minus one.

ARRAYS

An array is an ordered set of data in one, two, or three dimensions identified by a symbolic name. An array declarator (see *DIMENSION* statement) defines the name and size of the array. An array name serves to identify all of the elements in the array, including data type, real or integer.

## Array Element

An array element is one member of an array and is identified by a subscript appended to the array name.

## Subscripts

A subscript follows the array name and contains one, two, or three subscript expressions enclosed in parentheses. The number of subscript expressions (except in *EQUIVALENCE* statements) corresponds to the specified dimensionality of the array. Two or three expressions within the parentheses must be separated by a comma. Subscript expressions are type integer in one of the following forms:

$$c \pm v \pm k \qquad c \pm v \qquad v \pm k \qquad v \qquad k$$

where c and k are integer constants and v is an integer variable.

### Examples

$$X(2 \pm J - 3) \qquad A(I, J, K) \qquad B(20) \qquad C(L - 2)$$

## Dimensionality

Arrays are stored column-wise in ascending memory locations. Therefore, a two-dimensional real array, A, with three rows and three columns, would be stored internally in the computer as follows:

| Location               | Element |
|------------------------|---------|
| $L + 0 \ \& \ L + 1$   | A(1,1)  |
| $L + 2 \ \& \ L + 3$   | A(2,1)  |
| $L + 4 \ \& \ L + 5$   | A(3,1)  |
| $L + 6 \ \& \ L + 7$   | A(1,2)  |
| $L + 8 \ \& \ L + 9$   | A(2,2)  |
| .                      | .       |
| .                      | .       |
| .                      | .       |
| $L + 16 \ \& \ L + 17$ | A(3,3)  |

## data

The position of a real array element,  $A(i,j)$ , is derived from the following formula:

$$A^1 + (i-1 + 1 \times (j-1)) \times 2$$

where  $A^1$  is the location of the first element in the array,  $i$  and  $j$  are the specified row and column subscript expressions, and 1 is the number of row elements defined in the array declaration for  $A$ , and 2 is the number of words assigned to each real item. In the preceding example, the position of the  $A(2,2)$  element would be solved in the following form:

$$L + 0 + (2-1 + 3*(2-1)) \times 2 = L + 8$$

## SECTION 3 – SPECIFICATIONS AND STATEMENTS

### GENERAL

Specification statements organize and classify data that will be referred to by other statements in the **FORTRAN** program. Specification statements include:

|                    |                                                                                       |
|--------------------|---------------------------------------------------------------------------------------|
| <i>DIMENSION</i>   | Names and declares the size of an array.                                              |
| <i>COMMON</i>      | Assigns variable and/or named arrays to common storage areas.                         |
| <i>EQUIVALENCE</i> | Assigns variables and named arrays to shared storage areas.                           |
| <i>EXTERNAL</i>    | Names an external procedure                                                           |
| <i>TYPE</i>        | Declares entities to be of type integer, real, double precision, complex, or logical. |

Specification statements must precede all other statements except *BLOCK DATA*, *FUNCTION*, *SUBROUTINE*, and *NAME*.

### DIMENSION STATEMENT

Form: **DIMENSION** *V*<sub>1</sub>(*i*<sub>1</sub>), *V*<sub>2</sub>(*i*<sub>2</sub>), ..., *V*<sub>*n*</sub>(*i*<sub>*n*</sub>), where each *V*(*i*), (called an array declarator), is composed of a declarator name *V* (the name of the array), and a declarator subscript (*i*). Each (*i*) is an unsigned integer constant, two unsigned integer constants separated by a comma, or three unsigned integer constants separated by commas. Each constant must have a value greater than zero.

A *DIMENSION* statement specifies that the declarator names listed are arrays in the program unit. The number of dimensions and the maximum size of each dimension is specified by the declarator subscript associated with each declarator name.

More than one *DIMENSION* statement can appear in a program.

An array element is referred to by the array name qualified by a subscript to identify the desired element. If the value of this subscript is out of the range specified by the array declarator, the derived computational results will be unpredictable.

## statements

Array elements are stored column-wise in computer memory from low to high address storage. Therefore, one-dimension arrays are stored sequentially in the order  $A(1)$ ,  $A(2)$ ,.....,  $A(n)$ , while two-dimension arrays are stored with the first (leftmost) dimension varying most rapidly, i.e.,  $A(1,1)$ ,  $A(2,1)$ ,.....,  $A(m,1)$ ,  $A(1,2)$ ,  $A(2,2)$ ,.....,  $A(m,n)$ .

### Example

DIMENSION A(5), I1(3,6), C(5,10), BIG(10,10,10)

This specification statement indicates that A is a real vector with five elements; I1 is an integer matrix of size  $3 \times 6 = 18$  elements; C is a real matrix of size  $5 \times 10 = 50$  elements; and BIG is a real matrix of size  $10 \times 10 \times 10 = 1000$  elements.

## COMMON STATEMENT

General Form

COMMON /x/a,b,.../r/c,d,...

where:

a,b,...,c,d,... are variable or array names or array declarators.

/x/.../r/ represents optional common block names consisting of one through six alphanumeric characters, the first of which is alphabetic. These names must always be embedded in slashes.

Although the *COMMON* statement may be used to provide dimension information, adjustable dimensions may never be used.

Variables or arrays that appear in a calling program or subprogram may be made to share the same storage locations with variables or arrays in other subprograms by use of the *COMMON* statement. For example, if one program contains the statement:

COMMON TABLE

as its first *COMMON* statement, and a second program contains the statement:

COMMON TREE

as its first *COMMON* statement and the two programs are loaded together, the variable names TABLE and TREE refer to the same storage location.

If the main program contains the statement:

```
COMMON A, B, C
```

and a subprogram contains the statement:

```
COMMON X, Y, Z
```

then A shares the same storage location as X, B shares the same storage location as Y, and C shares the same storage location as Z.

Common entries appearing in *COMMON* statements are cumulative in the given order throughout the program; that is, they are cumulative in the sequence in which they appear in all *COMMON* statements. For example, consider the following two *COMMON* statements:

```
COMMON A, B, C
COMMON G, H
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H
```

Redundant entries are not allowed. For example, the following statement is invalid:

```
COMMON A, B, C, A
```

Consider the following example:

### Example

#### CALLING PROGRAM

```

      .
COMMON A, B, C, R(100)
INTEGER R
      .
      .
      .
CALL MAPMY (...)
```

#### SUBPROGRAM SUBROUTINE

```

MAPMY(...)
      .
      .
COMMON X, Y, Z, S(100)
INTEGER S
      .
      .
      .
```

## statements

### Explanation:

In the calling program, the statement `COMMON A, B, C, R(100)` would cause 206 storage locations (two locations per variable) to be reserved in the `COMMON` area.

The statement `COMMON X, Y, Z, S(100)` would then cause the variables `X, Y, Z, and S(1)...S(100)` to share the same storage spaces as `A, B, C, and R(1)...R(100)`, respectively.

From the above example, it can be seen that `COMMON` statements serve an important function: namely, as a medium to implicitly transmit data between the calling program and the subprogram. That is, values for `X, Y, Z, and S(1)...(100)`, because they occupy the same storage locations as `A, B, C, and R(1)...R(100)`, do not have to be transmitted in the argument list of a `CALL` statement. Arguments passed through `COMMON` must follow the same rules of presentation with regard to type, etc., as arguments passed in a list. (See the section entitled `SUBPROGRAMS`.)

## Blank and Labeled Common

In the preceding example, the common storage area (common block) established is called a blank common area. That is, no particular name was given to that area of storage. The variables that appeared in the `COMMON` statements were assigned locations relative to the beginning of the blank common area. However, variables and arrays may be placed in separate common areas. Each of these separate areas (or blocks) is given a name consisting of two through six alphanumeric characters (the first of which is alphabetic); those blocks which have the same name occupy the same storage space.

Those variables that are to be placed in labeled (or named) common are preceded by a common block name enclosed in slashes. For example, the variables `A, B, and C` will be placed in the labeled common area, `HOLD`, by the following statement:

```
COMMON/HOLD/A, B, C
```

In a `COMMON` statement, blank common can be distinguished from labeled common by preceding the variables in blank common by two consecutive slashes or, if the variables appear at the beginning of the common statement, by omitting any block name. For example, in the following statement:

```
COMMON A, B, C/ITEMS/X, Y, Z//D, E, F
```

the variables `A, B, C, D, E, and F` will be placed in blank common in that order; the variables `X, Y, and Z` will be placed in the `COMMON` area labeled `ITEMS`.

Blank and labeled common entries appearing in *COMMON* statements are cumulative throughout the program. For example, consider the following two *COMMON* statements:

```
COMMON A, B, C/R/D, E/S/F
COMMON G, H/S/I, J/R/P//W
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H, W/R/D, E, P/S/F, I, J
```

*COMMON* is allocated from low to high memory addresses.

## EQUIVALENCE STATEMENT

Form: *EQUIVALENCE* (*k*<sub>1</sub>), (*k*<sub>2</sub>),..., (*k*<sub>*n*</sub>), where each (*k*<sub>*i*</sub>) is a list of two or more nondummy variables and/or array element names, separated by commas. Subscript expressions of array element names must be nonzero, unsigned integer constants. An element of a two or three dimension array can be referred to by using a single subscript, giving the element position within the array (page 2-7).

The effect of the *EQUIVALENCE* statement is to cause the same area of memory to be shared by two or more entities. Each element of the *k<sub>i</sub>* list is assigned the same (or a part of the same) storage area.

More than one *EQUIVALENCE* statement is permitted in a program.

### Example

```
DIMENSION A(5), I1(3,3), B1(3)
COMMON B, B1, B2
EQUIVALENCE (X, A(2), Y), (B, C2, F5), (I1(5), B2)
```

The effect of an *EQUIVALENCE* statement upon *COMMON* assignments may be the lengthening of *COMMON*. This lengthening is permitted only if it increases *COMMON* in the same direction as additional *COMMON* elements would. Thus, in this example, the equivalence (B1(1), A(3)) would have been invalid. It is also invalid to equate two elements of the same array to each other.



statements

## TYPE STATEMENT

General Form:

TYPE a, b, ..., z

where:

TYPE is *INTEGER*, *REAL*, *DOUBLE PRECISION*, *COMPLEX*, or *LOGICAL*

a, b, ..., z represent variable or array names, or array declarators

The *TYPE* statements declare the type *INTEGER*, *REAL*, *DOUBLE PRECISION*, *COMPLEX* or *LOGICAL* of a particular variable or array by its name, rather than by its initial character. This differs from the other way of specifying the type of a variable or array (i.e., implicit type specification).

### Example 1

INTEGER ITEM, VALUE

Explanation:

This statement declares that the variables ITEM and VALUE are of type *INTEGER*.

### Example 2

REAL ARRAY, HOLD, VALUE, ITEM (5, 5)

Explanation:

This statement declares that the variables ARRAY, HOLD, VALUE, and the array named ITEM are of type *REAL*. In addition, it declares the size of the array ITEM. The variables ARRAY, HOLD, and VALUE have two storage locations reserved for each; and the array named ITEM has 50 storage locations reserved (two for each variable in the array).

### Example 3:

DOUBLE PRECISION C, D, E

Explanation:

This statement declares that the variables C, D, and E are of type *DOUBLE PRECISION*. Thus, C, D, and E each have four storage locations reserved, one for the exponent and three for the mantissa (page 2-3).

#### **Example 4**

**COMPLEX C, D, E**

Explanation:

This statement declares that the variables C, D, and E are of type **COMPLEX**. Thus C, D, and E each have four storage locations reserved (two for the real part, two for the imaginary part).



## SECTION 4 – EXPRESSIONS AND ASSIGNMENTS

### ARITHMETIC EXPRESSIONS

An arithmetic expression is formed in **FORTRAN** syntax by a combination of operators and elements. The expression and its elements identify the expression to be type *INTEGER*, *REAL*, *DOUBLE PRECISION*, or *COMPLEX*.

The arithmetic operators are:

| Operator | Representing   |
|----------|----------------|
| +        | addition       |
| -        | subtraction    |
| *        | multiplication |
| /        | division       |
| **       | exponentiation |

The arithmetic elements are described by the following statements:

|                                     |                                                                                                                                               |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <i>PRIMARY</i>                      | An <i>ARITHMETIC EXPRESSION</i> enclosed in parentheses, a constant, a variable reference, an array element reference, or function reference. |
| <i>FACTOR</i>                       | A <i>FACTOR</i> is a <i>PRIMARY</i> or a construct of the form: <i>PRIMARY**PRIMARY</i>                                                       |
| <i>TERM</i>                         | A <i>TERM</i> is a <i>FACTOR</i> or one of the forms:<br><i>TERM/FACTOR</i><br><i>TERM*TERM</i>                                               |
| <i>SIGNED TERM</i>                  | A <i>TERM</i> immediately preceded by a + or - sign.                                                                                          |
| <i>SIMPLE ARITHMETIC EXPRESSION</i> | A <i>TERM</i> or two <i>SIMPLE ARITHMETIC EXPRESSIONS</i> separated by a + or - sign.                                                         |

## arithmetic expressions

### ARITHMETIC EXPRESSION

A *SIMPLE ARITHMETIC EXPRESSION* or a signed *TERM* or either of the preceding immediately followed by a + or - sign and a *SIMPLE ARITHMETIC EXPRESSION*.

A *PRIMARY* of any type may be exponentiated by an *INTEGER PRIMARY* and the resulting factor is of the same type as that of the element being exponentiated. A *REAL* or *DOUBLE-PRECISION PRIMARY* may be exponentiated by a *REAL* or *DOUBLE-PRECISION PRIMARY*. The resultant *FACTOR* is of type *REAL* if both *PRIMARIES* are *REAL*, and otherwise of type *DOUBLE PRECISION*. These are the only cases for which use of the exponentiation operator is defined. Valid combinations for exponentiation are:

| Base                    | Exponent                                                                        |
|-------------------------|---------------------------------------------------------------------------------|
| <i>REAL</i> ,           | <i>REAL</i> , <i>INTEGER</i> or <i>DOUBLE PRECISION</i>                         |
| <i>INTEGER</i>          | <i>INTEGER</i> ( <i>REAL</i> and <i>DOUBLE PRECISION</i> exponents are invalid) |
| <i>DOUBLE PRECISION</i> | <i>REAL</i> , <i>INTEGER</i> or <i>DOUBLE PRECISION</i>                         |

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type.

Further, an admissible real element may be combined with an admissible double-precision or complex element; the resultant element is of type *DOUBLE PRECISION* or *COMPLEX*, respectively.

A part of an expression is evaluated only if it is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. The range of the subtraction operator is the term of the operator that immediately succeeds it. The evaluation may proceed according to any valid formation sequence. Use of an array element name requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript. An element whose value is not mathematically defined cannot be evaluated.

The following rules represent the derivation of all permissible expressions:

A variable, constant, or function standing alone is an expression.

A(1)  
JOBNO  
217  
17.26  
SQRT(A + B)

If E is an expression whose first character is not an operator, then +E and -E are expressions.

-A(1)  
+JOBNO  
-217  
+17.26  
-SQRT(A + B)

If E is an expression, then (E) is an expression meaning the quantity E taken as a unit.

(-A)  
-( + JOBNO)  
-(X + Y)  
(A-SQRT(A + B))

If E is an expression whose first character is not an operator, and F is an expression, then: F + E, F - E, F \* E, F / E and F \*\* E are all expressions.

-(B(I,J) + SQRT(A + B(K,L)))  
-(B(I + E, 3 \* J + K) + A)  
1.7E-2 \*\* (X + 5.0)

The mode of expression is determined by the modes of its elements, which must be the same with the following exceptions:

- a. A *REAL* quantity can appear in an *INTEGER* expression only as an argument of a function.  
I + LFUNC(B)
- b. An *INTEGER* quantity can appear in a *REAL* expression only as an argument of a function, or as a subscript or an exponent.  
AFUNC(I + 2)  
A(I, J + 1)  
B \*\* N

The order of evaluation of expressions is established by the use of parentheses in the statement. If parentheses are not indicated, the following conventions of mathematics apply.

The hierarchy of operations, in order of precedence is: exponentiation, followed by multiplication and division, followed by addition and subtraction.

assignments

Within the same hierarchy of operations, evaluation proceeds from left to right.

Examples

|                      |                   |                             |
|----------------------|-------------------|-----------------------------|
| $X + Y * Z$          | is interpreted as | $X + (Y * Z)$               |
| $W * X / Y * Z$      | is interpreted as | $((W * X) / Y) * Z$         |
| $B ** 2 - 4 * A * C$ | is interpreted as | $(B ** 2) - ((4. * A) * C)$ |
| $X \cdot Y \cdot Z$  | is interpreted as | $(X \cdot Y) \cdot Z$       |
| $X / Y / Z$          | is interpreted as | $(X / Y) / Z$               |

ARITHMETIC ASSIGNMENT STATEMENT

General Form

$a = b$

where:

a is any subscripted or unsubscripted variable.

b is any arithmetic expression.

This **FORTRAN** statement closely resembles a conventional algebraic equation; however, the equal sign specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign.

Rules for Assignment of b to a

| If a is TYPE     | and b is TYPE    | the assignment rule is*     |
|------------------|------------------|-----------------------------|
| INTEGER          | INTEGER          | Assign                      |
| INTEGER          | REAL             | Fix and Assign              |
| INTEGER          | DOUBLE PRECISION | Fix and Assign              |
| INTEGER          | COMPLEX          | P                           |
| REAL             | INTEGER          | Float and Assign            |
| REAL             | REAL             | Assign                      |
| REAL             | DOUBLE PRECISION | DP Evaluate and Real Assign |
| REAL             | COMPLEX          | P                           |
| DOUBLE PRECISION | INTEGER          | DP Float and Assign         |
| DOUBLE PRECISION | REAL             | DP Evaluate and Assign      |
| DOUBLE PRECISION | DOUBLE PRECISION | Assign                      |
| DOUBLE PRECISION | COMPLEX          | P                           |

|         |                  |        |
|---------|------------------|--------|
| COMPLEX | INTEGER          | P      |
| COMPLEX | REAL             | P      |
| COMPLEX | DOUBLE PRECISION | P      |
| COMPLEX | COMPLEX          | Assign |

\*Notes

|               |                                                                                                            |
|---------------|------------------------------------------------------------------------------------------------------------|
| P =           | prohibited combination                                                                                     |
| Assign =      | transmit the resulting value without change                                                                |
| Real assign = | transmit as much precision of the most significant part of the resulting value as a REAL datum can contain |
| DP evaluate = | evaluate according to the most precise rules, then DP float                                                |
| Fix =         | truncate any fractional part and transform to INTEGER                                                      |
| Float =       | transform to REAL                                                                                          |
| DP float =    | transform to DOUBLE PRECISION retaining as much precision as a DOUBLE PRECISION datum can contain          |

Assume that the type of the following variables has been specified as:

| Variable Names | Type              |
|----------------|-------------------|
| I, J, W        | INTEGER variables |
| A, B, C, D     | REAL variables    |
| E              | COMPLEX variable  |

Then the following examples illustrate valid arithmetic statements using constants, variables, and subscripted variables of different types:

| Statement | Description                                                                              |
|-----------|------------------------------------------------------------------------------------------|
| A = B     | The value of A is replaced by the current value of B.                                    |
| W = B     | The value of B is truncated to an integer value, and this value replaces the value of W. |
| *A = I    | The value of I is converted to a real value, and this result replaces the value of A.    |



## assignments

|                  |                                                                                                                                                                        |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $I = I + 1$      | The value of I is replaced by the value of $I + 1$ .                                                                                                                   |
| $A = C * D$      | The most significant part of the product of C and D replaces the value of A.                                                                                           |
| $E = (1.0, 2.0)$ | The value of the complex variable E is replaced by the complex constant (1.0, 2.0). Note that the statement $E = (A, B)$ where A and B are real variables, is invalid. |

## LOGICAL ASSIGNMENT STATEMENT

General Form

$$a = b$$

where:

a is a subscripted or nonsubscripted variable.

b is any logical expression.

| Variable Names | Type              |
|----------------|-------------------|
| G, H           | LOGICAL variables |

Examples of logical assignment statements are:

| Statement     | Description                                                                                                                                                                                                                           |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $G = .TRUE.$  | The value of G is replaced by the logical constant .TRUE..                                                                                                                                                                            |
| $H = .NOT.G$  | If G is .TRUE., the value of H is replaced by the logical constant .FALSE.. If G is .FALSE., the value of H is replaced by the logical constant .TRUE..                                                                               |
| $G = 3..GT.I$ | The value of I is converted to a real value; if the real constant 3. is greater than this result, the logical constant .TRUE. replaces value of G. If 3. is not greater than I, the logical constant .FALSE. replaces the value of G. |

## LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical constant, logical variable, or logical subscripted variable, the value of which is always a truth value (i.e., either `.TRUE.` or `.FALSE.`).

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of the three following forms:

- a. Relational operators combined with arithmetic expressions whose mode is *INTEGER*, *REAL*, or *DOUBLE PRECISION*.
- b. Logical operators combined with logical constants (`.TRUE.` and `.FALSE.`), logical variables, or subscripted logical variables.
- c. Logical operators combined with either or both forms of the logical expressions described in items a and b.

Item a is discussed in the following section, Relational Operators; items b and c are discussed in the section entitled Logical Operators.

### Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator and has the value `.TRUE.` or `.FALSE.` as the relation is true or false. Both arithmetic expressions can be type *INTEGER* or one may be type *REAL* or *DOUBLE PRECISION* and the other type *REAL* or *DOUBLE PRECISION*. If a real and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right-hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged.

The six relational operators, each of which must be preceded and followed by a period, are as follows:

| Relational Operator | Definition                          |
|---------------------|-------------------------------------|
| <code>.GT.</code>   | Greater than ( $>$ )                |
| <code>.GE.</code>   | Greater than or equal to ( $\geq$ ) |
| <code>.LT.</code>   | Less than ( $<$ )                   |
| <code>.LE.</code>   | Less than or equal to ( $\leq$ )    |
| <code>.EQ.</code>   | Equal to ( $=$ )                    |
| <code>.NE.</code>   | Not equal to ( $\neq$ )             |

logical expressions

The relational operators express an arithmetic condition which can be either true or false. Only arithmetic expressions whose mode is *INTEGER*, *REAL*, or *DOUBLE PRECISION* can be combined by relational operators. For example, assuming the type of variable has been specified as follows:

| Variable Names | Type                     |
|----------------|--------------------------|
| ROOT, E, Q     | <i>REAL</i> variables    |
| A, I, F        | <i>INTEGER</i> variables |
| L              | <i>LOGICAL</i> variable  |
| C              | <i>COMPLEX</i> variable  |

then, the following illustrates valid and invalid logical expressions using the relational operators.

Example

Valid Logical Expressions Using Relational Operators:

(ROOT\*Q).GT.E  
A.LT.I  
E\*\*2.7.EQ.(5.\*ROOT + 4.)  
57.9.LE.(4.7 + E)  
.5.GE.9.\*ROOT  
E.EQ.27.3E + 05

Invalid Logical Expressions Using Relational Operators:

|                  |                                                                 |
|------------------|-----------------------------------------------------------------|
| C.LT.ROOT        | Complex quantities can never appear in logical expressions.     |
| C.GE.(2.7,5.9E3) | Complex quantities can never appear in logical expressions.     |
| L.EQ.(A + F)     | Logical quantities may never be joined by relational operators. |
| E**2.EQ97.1E9    | Missing period immediately after the relational operator.       |
| .GT.9            | Missing arithmetic expression before the relational operator.   |

## Logical Operators

The three logical operators, each of which must be preceded and followed by a period, are as follows: (A and B represent logical constants or variables, or expressions containing relational operators).

| Logical Operator | Definition                                                                                                                                        |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| .NOT.            | .NOT.A - if A is .TRUE., then .NOT.A has the value .FALSE.; if A is .FALSE., then .NOT.A has the value .TRUE.                                     |
| .AND.            | A.AND.B - if A and B are both .TRUE., then A.AND.B has the value .TRUE.; if either A or B or both are .FALSE., then A.AND.B has the value .FALSE. |
| .OR.             | A.OR.B - if either A or B - or both are .TRUE., then A.OR.B has the value .TRUE.; if both A and B are .FALSE., then A.OR.B has the value .FALSE.  |

Two logical operators may appear in sequence only if the second one is the logical operator .NOT..

Only those expressions which, when evaluated, have the value .TRUE. or .FALSE. may be combined with the logical operators to form logical expressions. For example, assume that the type of variable has been specified as follows:

| Variable Names | Type              |
|----------------|-------------------|
| ROOT, E, Q     | REAL variables    |
| A, I, F        | INTEGER variables |
| L, W           | LOGICAL variables |
| C              | COMPLEX variable  |

Then the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

## logical operators

### Examples

Valid Logical Expressions:

`(ROOT**Q.GT.L).AND.W`

`L.AND..NOT.(I.GT.F)`

`(E + 5.9E2.GT.2.*E).OR.L`

`.NOT.W.AND..NOT.L`

`L.AND..NOT.W.OR.I.GT.F`

`(E**F.GT.ROOT).AND..NOT.(I.EQ.A)`

### Invalid Logical Expressions

`A.AND.L`

A is not a logical expression.

`.OR.W`

.OR. must be preceded by a logical expression.

`NOT.(A.GT:F)`

missing period before the logical operator  
.NOT.

`(C.EQ.I).AND.L`

a complex variable may never appear in a  
logical expression.

`L.AND..OR.W`

the logical operators .AND. and .OR. must  
always be separated by a logical expression.

`.AND.L`

.AND. must be preceded by a logical expres-  
sion.

### Order of Computations in Logical Expressions

Where parentheses are omitted, or where the entire logical expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

| Operation                             | Hierarchy     |
|---------------------------------------|---------------|
| Evaluation of Functions               | 1st (highest) |
| Exponentiation (**)                   | 2nd           |
| Multiplication and division (* and /) | 3rd           |
| Addition and subtraction (+ and -)    | 4th           |
| .LT.,.LE.,.EQ.,.NE.,.GT.,.GE.         | 5th           |
| .NOT.                                 | 6th           |
| .AND.                                 | 7th           |
| .OR.                                  | 8th           |

For example, the expression:

`(A.GT.D**B.AND..NOT.L.OR.N)`

is effectively evaluated in the following order.

- |            |                                                     |
|------------|-----------------------------------------------------|
| 1. D**B    | Call the result W (exponentiation)                  |
| 2. A.GT.W  | Call the result X (relational operator)             |
| 3. .NOT.L  | Call the result Y (highest logical operator)        |
| 4. X.AND.Y | Call the result Z (second highest logical operator) |
| 5. Z.OR.N  | Final operation                                     |

### Use of Parentheses in Logical Expressions

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the most deeply nested parentheses (that is, the innermost pair of parentheses) is effectively evaluated first. For example, the logical expression:

`((I.GT.(B + C)).AND.L)`

is effectively evaluated in the following order.

- |            |                   |
|------------|-------------------|
| 1. B + C   | Call the result X |
| 2. I.GT.X  | Call the result Y |
| 3. Y.AND.L | Final operation   |

The logical expression to which the logical operator `.NOT.` applies must be enclosed in parentheses if it contains two or more quantities. For example, assume that the values of the logical variables, A and B, are `.FALSE.` and `.TRUE.`, respectively. Then the following two expressions are not equivalent:

`.NOT.(A.OR.B)`  
`.NOT.A.OR.B`

In the first expression, A.OR.B is evaluated first. The result is .TRUE., but .NOT. (.TRUE.) implies .FALSE.. Therefore, the value of the first expression is .FALSE..

In the second expression, .NOT.A is evaluated first. The result is .TRUE.; but .TRUE.OR.B implies .TRUE.. Therefore, the value of the second expression is .TRUE..

## SECTION 5 – CONTROL STATEMENTS

### GENERAL

Each statement in a **FORTRAN** program is executed in the order of its appearance in the source program, unless this sequence is interrupted or modified by a control statement. This section of the manual describes the control statements used in **FORTRAN**.

### GO TO STATEMENTS

*GO TO* statements transfer logical control from one section of a program to another. **FORTRAN** includes three forms of the *GO TO* statement: unconditional, computed, and assigned *GO TO*.

#### Unconditional GO TO

An unconditional *GO TO* is of the form: *GO TO k*, where *k* is a statement label reference.

Execution of this statement causes the statement identified by the label *k* to be executed next in sequence.

#### Example

```
GO TO 72  
.  
71 V7 = HQ(5) + Y**L  
.  
.  
.  
72 V7 = HQ(4) + X**J
```

In this example, execution of the *GO TO 72* statement causes statement number 71 and any succeeding statements to be bypassed. Execution is resumed with statement number 72.

#### Computed GO TO

The computed *GO TO* statement is of the form: *GO TO (k1, k2, ..., kn), i*, where the *k*'s are statement label references, and *i* is an integer variable reference.

Execution of this statement causes the statement identified by the statement label *kj* to be executed next in sequence, where *j* is the value of *i* at execution time. Valid execution



## control statements

of this statement is dependent upon the value of the integer variable such that 1 is less than or equal to j, and j is less than or equal to n.

### Example

GO TO (98, 405, 3), n

Execution of the statement in the example will cause control to be transferred to the statement labeled 98, 405, or 3 if the value of the variable integer n is 1, 2, or 3, respectively.

## ASSIGN and Assigned GO TO

General Form

```
ASSIGN i TO m
      .
      .
      .
GO TO m, (X1, X2, X3,...,Xn)
```

where:

i is an executable statement number.

X1,X2,X3,...,Xn are executable statement numbers.

m is a nonsubscripted integer variable that is assigned one of the following statement numbers: X1, X2, X3,...Xn.

The assigned GO TO statement causes control to be transferred to the statement numbered X1, X2, X3,..., or Xn, depending on whether the current assignment of m is X1, X2, X3,..., or Xn, respectively. For example, in the following statement:

GO TO N, (10, 25, 8)

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed next. If the current assignment of N is statement number 10, the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next.

The current assignment of the integer variable m is determined by the last executed ASSIGN statement. Only an ASSIGN statement may be used to initialize or change the value of m. The value of m is not the integer statement number; ASSiGN 10 TO l is not the same as l = 10.

**Example 1**

```

      .
      .
      .
      ASSIGN 50 TO NUMBER
      .
10  GO TO NUMBER, (35, 50, 25, 12, 18)
      .
      .
      .
50  A = B + C
      .
      .
      .

```

In the above example, statement 50 is executed immediately after statement 10.

**Example 2**

```

      .
      .
      .
      ASSIGN 10 TO ITEM
      .
      .
      .
13  GO TO ITEM, (8, 12, 25, 50, 10)
      .
      .
      .
8   A = B + C
      .
      .
      .
10  B = C + D
      .
      ASSIGN 25 TO ITEM
      GO TO 13
      .
      .
      .
25  C = E ** 2
      .
      .
      .

```

## control statements

In the above example, the first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

## ARITHMETIC IF STATEMENT

It is often necessary to alter the logical flow of a program on the basis of the results of an arithmetic test. The *IF* statement is a conditional transfer that will execute this level of control, and is of the form:

IF (e) k1, k2, k3

The arithmetic *IF* is a three-way transfer. Execution of this statement causes the expression (e) to be evaluated, following which the statement identified by the label k1, k2, k3 is executed next in sequence, as the value of (e) is less than zero, equal to zero, or greater than zero, respectively.

### Example

```
IF (I) 10, 11, 12
10 V7 = HQ(5) + Y**L

GO TO 13
11 V7 = HQ(4) + X**J

GO TO 13
12 V7 = HQ(3) + X**L

13 NEXT STATEMENT

5-4
```

In this example, execution of IF (I) 10, 11, 12 causes one of the following actions: for a negative value of I, statement number 10 is executed in sequence; for a zero value of I, statement number 10 and any succeeding statements are bypassed and statement number 11 is executed; for a positive nonzero value of I, statements 10 through 11 and any statement following statement 11 are bypassed, and statement number 12 is executed.

## Logical IF Statement

General Form

IF (a)s

where:

a is any logical expression.

s is any executable statement except a *DO* statement  
or another logical *IF* statement

The logical *IF* statement is used to evaluate the logical expression (a) and to execute or skip statement s depending on whether the value of the expression is *.TRUE.* or *.FALSE.*, respectively.

#### Example

```

      .
      .
      .
      5 IF (A.LE.0.0) GO TO 25
      10 C = D + E
      15 IF (A.EQ.B) ANSWER = 2.0*A/C
      20 F = G/H
      .
      .
      .
      25 W = X**Z
      .
      .
      .

```

In statement 5, if the value of the expression is *.TRUE.* (i.e., A is less than or equal to 0.0), the statement *GO TO 25* is executed next and control is passed to the statement numbered 25. If the value of the expression is *.FALSE.* (i.e., A is greater than 0.0), the statement *GO TO 25* is ignored and control is passed to the statement numbered 10.

In statement 15, if the value of the expression is *.TRUE.* (i.e., A is equal to B), the value of *ANSWER* is replaced by the value of the expression  $(2.0 * A / C)$ , and statement 20 is executed. If the value of the expression is *.FALSE.* (i.e., A is not equal to B), the value of *ANSWER* remains unchanged and statement 20 is executed next.

#### Example

```

      .
      .
      .
      5 IF (P.OR..NOT.Q) A = B
      10 C = B**2
      .
      .
      .

```

## control statements

Assume that P and Q are logical variables. In statement 5, if the value of the expression is .TRUE., the value of A is replaced by the value of B and statement 10 is executed next. If the value of the expression is .FALSE., statement A = B is skipped and statement 10 is executed.

## CALL STATEMENT

The *CALL* statement causes a transfer of execution control to a subroutine-type subprogram, and is of one of the forms: *CALL s(a1, a2,..., an)* and *CALL s*, where *s* is the name of a subroutine and the *a*'s are actual arguments that will replace the dummy arguments in the called subroutine. Arguments can be Hollerith constants, variable names, array element names, array names, any other expression, or the name of an external procedure. They must, however (except for Hollerith constants), be indicated in order, number, and type with the corresponding dummy arguments of the subroutine.

Execution of the *CALL* statement transfers control to the designated subroutine. The arguments declared in the statement line are associated with the dummy arguments that are parameters of the executable statements of the subroutine. Control is then passed to the first executable statement of the called subroutine. Control will be returned to the first executable statement following the *CALL* statement upon execution of the *RETURN* statement in the subroutine. Examples of calling sequences to subroutines are shown below.

```
CALL TEST (A,I)  
CALL EXIT
```

The first example will transfer execution control to the subroutine labelled *TEST* and include the parameters or arguments *A* and *I* in the subroutine. The second example will cause execution control to be transferred to the subroutine labelled *EXIT*. Any arguments required for execution of *exit* are self-contained in the logic of the subroutine.

## RETURN STATEMENT

The execution of a *RETURN* statement results in the exit from a subprogram, and is expressed in the form: *RETURN*. A *RETURN* statement defines the logical end of a procedure subprogram and, therefore, may appear only in a subprogram. Execution of the statement returns logical control to the current calling program unit. Each subprogram must contain at least one *RETURN* statement.

In the case of a subroutine subprogram, control is returned to the first statement immediately following the *CALL* statement that released control to the subroutine. In the case of a function subprogram, control is returned (with the value of the function available) to the statement that called the function subprogram.

## CONTINUE STATEMENT

Form: CONTINUE.

The *CONTINUE* statement results in no action in an execution sequence; therefore, the statement has no effect upon the program. This statement serves as a program unit reference point and is frequently used at the end of a *DO* loop.

### Example

```

      IF (I) 10, 11, 12
10  V7 = HQ(5) + Y**L
      .
      .
      GO TO 13
11  V7 = HQ(4) + X**J
      .
      .
      GO TO 13
12  V7 = HQ(3) + X**L
      .
      .
13  CONTINUE

```

## PAUSE STATEMENT

Form: PAUSE n or PAUSE, where n is an octal digit string of length from one to five digits designating the particular *PAUSE*.

A *PAUSE* statement causes a temporary cessation of program execution and displays PAUSE n (see section 8 for display format). The statement permits operator intervention for setup or control functions, such as changing data tapes. The computer executes a halt instruction, delaying further execution until the operator selects the console RUN button. Execution will resume at the first executable statement following the *PAUSE* statement.

### Example

```

PAUSE 01

```

## control statements

### STOP STATEMENT

Form: STOP *n* or STOP, where *n* is an octal digit string of length from one to five digits designating the particular STOP.

A STOP statement causes termination of program execution and displays STOP *n* (see section 8 for display format). The program then terminates.

#### Example

STOP 0721

### DO STATEMENT

The DO statement controls repetitive execution of a group of statements. The number of repetitions depends on the value of a control variable. The statement assumes one of the forms: DO *n i* = *m1*, *m2*, *m3* and DO *n i* = *m1*, *m2*, where *n* is the statement label of an executable statement. This statement, called the terminal statement of the associated DO must physically follow and be in the same program unit as the DO statement. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, or another DO statement, nor a logical IF statement containing one of these forms.

Symbol *i* is an integer variable name, identified as the control variable.

Symbol *m1*, identified as the initial parameter, *m2*, as the terminal parameter, and *m3*, as the incrementation parameter are each either an integer constant or integer variable reference. If the second form of the DO statement is used, a value of 1 is implied for the incrementation parameter. When the DO statement is executed, the values of *m1*, *m2*, and *m3* must be greater than zero.

Associated with each DO statement is a range that is defined to be those executable statements from and including the first executable statement following the DO, to and including the terminal statement defined by the DO. A special situation occurs when the range of a DO contains another DO statement. In this case, the range of the contained DO must be a subset of the range of the containing DO.

The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

The range of the DO is executed.

If control reaches the terminal statement after execution of the terminal statement, the control variable of the most recently executed DO statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

If the value of the control variable is greater than the value represented by its associated terminal parameter, the *DO* is said to be satisfied, and the control variable becomes undefined.

If there were one or more other *DO* statements referring to the terminal statements in question, the control variable of the next most recently executed *DO* statement is incremented by the value represented by the associated incrementation parameter until all *DO* statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a *DO* by execution of a *GO TO* statement or an arithmetic *IF* statement, that is other than by satisfying the *DO*, the control variable of the *DO* is defined and is equal to the most recent attained value.

A *GO TO* or arithmetic *IF* statement may not cause control to pass into the range of a *DO* from outside its range. When a procedure reference occurs in the range of a *DO*, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial, terminal, and incrementation parameters of a *DO* may not be redefined during the execution of the range of that *DO*.

If a statement is the terminal statement of more than one *DO* statement, the label of that terminal statement may not be used in any *GO TO* or arithmetic *IF* statement that occurs anywhere but in the range of the most deeply contained *DO* with that terminal statement.

#### Example

DO 607 K1=2, ID, 3

The foregoing statement would cause K1, the control variable, to be set to the value of the initial parameter, 2. Execution would proceed at the statement immediately following, down to and including the statement identified by the label 607. After each execution of the loop, K1 is incremented by the incrementation parameter, 3, and evaluated in relation to the current value of the terminal parameter, ID. If the current value of K1 = ID, execution control is transferred to the statement following that identified by the label 607; otherwise, the *DO* cycle is repeated.

Example illustrating *DO* nesting:

```

WRITE (MX,8)
L = 0
DO 150 J = 1,K
DO 140 I = 1,M
L = L + 1
140 D(I) = V(L)

```



## control statements.

```
150  WRITE (MX,9)J,(D(I),I = 1,M)
      CALL LOAD (M,K,R,V)
C    PRINT FACTOR MATRIX
      WRITE (MX,10)K
      DO 180 I = 1,M
      DO 170 J = 1,K
        L = M*(J-1) + I
170    D(J) = V(L)
180    WRITE (MX,11)I,(D(J),J = 1,K)
        IF (K-1) 185, 185, 188
185    WRITE (MX,19)K
        GO TO 100
188    CALL VARMX (M,K,V,NC,TV,B,T,D)
```

## SECTION 6 — INPUT/OUTPUT STATEMENTS

### GENERAL

Input statements provide a program with the means of receiving information from external sources. Output statements allow the transmission of program data to external sources. These external sources may be devices such as magnetic tape and paper tape handlers, typewriters, and punch card processors.

There are two types of input/output statements. *READ* and *WRITE* statements  
*AUXILIARY* Input/Output statements

The first statement type causes the transfer of records of sequential files to and from the program. These data may be formatted information consisting of strings of characters or unformatted information consisting of binary word values in the form in which they normally appear in storage. The second statement type consists of the *BACKSPACE* and *REWIND* statements, which provide for positioning of devices and the *ENDFILE* statement, which provides for writing an end of file indicator.

Input/output statements reference input/output units, formatted information, and format specifications. An input/output unit is identified by a logical unit number, *u*, which can be an integer constant or a variable name referencing an integer constant. Logical unit number assignments for **FORTRAN** are listed in section 8. The format specification *f* is defined by either a **FORMAT** statement having the statement label *f*, or an array name. If *f* is a **FORMAT** statement label, the statement must appear in the same program as the input/output statement.

### INPUT/OUTPUT LISTS

The input list specifies the names of variables and array elements to which input values are assigned. The output list specifies the names of variables and array elements whose values are to be transmitted. Input and output lists are of the same form.

### SIMPLE LISTS

Simple lists have the form: *m*<sub>1</sub>, *m*<sub>2</sub>, *m*<sub>3</sub>..., *m*<sub>*n*</sub>, where *m*<sub>*i*</sub> is the name of a variable or array element. Commas separate each name in the list. The period signifies possible additional list items. List elements can be enclosed in parentheses.

| Example | Input Lists     | Output Lists       |
|---------|-----------------|--------------------|
|         | A               | B                  |
|         | C(26,L)         | I(10,10)           |
|         | R, K, D, (I, J) | S, (R, K), F(1,25) |

## I/O statements

An array variable in a list that is not subscripted is considered equivalent to the listing of each successive element of the array. If B is an array, list B is equivalent to B (1, 1), B (2, 1), B (3, 1),..., B (1, 2), B (2, 2),..., B (j, k), where j and k are the subscript limits of B.

## DO-IMPLIED LISTS

A DO-implied list is a simple list followed by a comma character and an expression of the form:  $i = m_1, m_2, m_3$  or  $i = m_1, m_2$ .

The elements i, m<sub>1</sub>, m<sub>2</sub>, and m<sub>3</sub> have the same meaning as defined for the DO statement. The DO implication applies to all simple list items enclosed in parentheses with the implication. For input lists, i, m<sub>1</sub>, m<sub>2</sub>, and m<sub>3</sub> may appear within this range only as subscripts.

### Examples

DO-Implied Lists:

```
(X (I), I = 1, 4)
(Q (J), R (J), J = 1, 2)
(G (K), K = 1, 7, 3)
((A (I, J), I = 3, 5), J = 1, 2)
(X (K), K = 1, 2), I, (R (J), J = 3, 5)
```

Equivalent Simple Lists:

```
X (1), X (2), X (3), X (4)
Q (1), R (1), Q (2), R (2)
G (1), G (4), G (7)
A (3, 1), A (4, 1), A (5, 1), A (3, 2), A (4, 2), A (5, 2)
X (1), X (2), I, R (3), R (4), R (5)
```

## READ STATEMENTS

These statements are used to obtain data values from an external source. The data values are input in either formatted or unformatted mode. The form of a formatted READ statement is:

READ (u,f) k.

The verb READ and the parentheses must appear in this form.

Execution of this statement causes information to be transmitted from the external source whose logical unit number is defined by u (section 8 and appendix A). These data are scanned and converted as specified by the format specification, f, and the resulting values are assigned to the variable names defined in the list, k.

The form of an unformatted *READ* statement is: *READ* (u) k.

The verb *READ* and the parentheses must appear in this form.

This statement causes data to be input in binary form from the unit defined by u. The values are assigned to the variable names defined in the list, k.

### Examples

```
READ (1,44) A, B, C
READ (2) R, S
READ (N, 12) A, (R (I), I = 1, 10)
READ (L) S, (T (J), J = 1, N)
```

All information appearing on external sources is divided into records. Each time a *READ* statement is executed, a new record is processed. The number of records input by a single *READ* statement is determined by the list and format specification. If only part of a record is input the remainder of the record is lost as the next *READ* processes the next record. Records are read sequentially until the list is exhausted. Only enough values are read to fill the list.

The list, k, in an unformatted *READ* statement may be left blank to skip a record.

Formatted and unformatted records are described in sections 8.

## WRITE STATEMENTS

*WRITE* statements are used to transfer program data to external devices. These data may be formatted or unformatted. The form of a formatted *WRITE* statement is: *WRITE* (u, f) k.

The verb *WRITE* and the parentheses must appear in this form.

Execution of this statement causes records to be written on the device referenced by u. The contents of the records are the values taken sequentially from the list, k, converted according to the format specification, f.

The form of an unformatted *WRITE* statement is:

```
WRITE (u) k.
```

The verb *WRITE* and the parentheses must appear in this form.

Execution of this statement causes binary information from the list, k, to be written in records on the unit defined by u.

### Examples

```
WRITE (1, 4) A, B, C  
WRITE (7) R, S, T  
WRITE (K, 12) X, (Y (J), J = 1, M), I  
WRITE (N) W, Z, (F (K), K = 1, 5)
```

Several records may be written with a single *WRITE* statement. The number of records is determined by the list and format specifications. Successive records are written until the data are exhausted. If the data do not fill a record, the record is filled with blanks.

## REWIND STATEMENT

This statement is of the form:

```
REWIND u.
```

Execution of this statement causes the physical unit defined by *u* to be rewound.

## BACKSPACE STATEMENT

This statement has the form:

```
BACKSPACE u.
```

The *BACKSPACE* statement causes the physical unit defined by *u* to be backspaced one record.

## ENDFILE STATEMENT

This statement has the form:

```
ENDFILE u.
```

When this statement is executed, an end of file indicator is written on the physical unit defined by *u*.

## FORMAT STATEMENTS

*FORMAT* statements, with input/output operations, specify conversion and editing of information between program storage and external representation. *FORMAT* statements are nonexecutable and must have a statement label to be referenced by input/output statements. Conversion performed according to a *FORMAT* statement during output is in general the reverse of conversion performed during an input operation.

A *FORMAT* statement is expressed as:

n *FORMAT* (f1, f2, f3, ..., fn)

where

n is the statement label and the fn are field specifications

The noun *FORMAT* and the parentheses must appear in this form.

When formatted records are output to a printer, the first character of the record is not printed but is processed as a printer vertical spacing control character as follows:

| Character | Vertical Spacing           |
|-----------|----------------------------|
| blank     | One line                   |
| 0         | Two lines                  |
| 1         | To first line of next page |

## FIELD SPECIFICATIONS

*FIELD* specifications describe the type of conversion and editing to be performed on each variable appearing in the input/output list. *FIELD* specifications can be in any of the following forms:

rAw rFw.d rEw.d rDw.d rlw nHs nX rLw rGw.d

where:

- The characters A, D, E, F, G, L, and I indicate the manner of conversion for variables in the list.
- The characters H and X represent characters to be input/ output directly from the format.
- The character / represents the end of a record.
- w and n are nonzero integer constants defining the width of the field (including digits, decimal points, and algebraic signs) in the external character string.
- d is an integer specifying the number of fractional digits appearing in the external string.
- r is an optional, nonzero integer indicating that the specification is to be repeated r times.
- s is a string of acceptable **FORTRAN** characters.

**F CONVERSION**

Form

rFw.d

Only real data may be processed by this form of conversion.

**Output**

The field is right-justified with as many leading blanks as necessary to fill w. Negative values are preceded by a minus sign. Internal values are converted to fixed-point decimal numbers and rounded to d decimal places.

For a field specification of F10.4:

|           |                 |          |
|-----------|-----------------|----------|
| 368.4     | is converted to | 368.4000 |
| 12.0      | is converted to | 12.0000  |
| -17.90767 | is converted to | -17.9077 |
| 37.5E-2   | is converted to | .3750    |

If a value requires more positions than allowed by w, the most significant digits, including sign if negative, are output. The error indication is designated by an asterisk in the least significant character position.

For a field specification of F6.4:

|         |                 |        |
|---------|-----------------|--------|
| 4739.76 | is converted to | 4740*  |
| -12.463 | is converted to | -12.5* |

**Input**

Input strings are decimal numbers of length w with d characters in the fractional portion. Blanks are treated as zeros. If a decimal point is present in a value, the fractional portion of the value is explicitly defined by that decimal point character.

For a field specification F8.3:

|         |                 |         |
|---------|-----------------|---------|
| 35      | is converted to | 0.035   |
| 964372  | is converted to | 964.372 |
| 0.53821 | is converted to | 0.53821 |
| -16.402 | is converted to | -16.402 |
| -12     | is converted to | -0.012  |
| 47.E-4  | is converted to | 0.0047  |

## E CONVERSION

Form

rEw.d

Only real data may be processed by this form of conversion.

### Output

Internal values are converted to decimal values of the forms:

.ddd...dE + ee and .ddd...E-ee

where:

ddd...d represents d digits, and  
ee is a decimal exponent.

The leading decimal point and E characters are present exactly as shown. Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right-justified and preceded by blanks to fill the width, w. This field width includes the exponent digits, the sign of the exponent (minus or space), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or space). This means that the field width should correspond to the relation:  $w \geq d + 6$ .

If w is less than (d + 6), the format is in error.

For a field specification of E12.5:

|            |                 |             |
|------------|-----------------|-------------|
| 76.573     | is converted to | .76573E 02  |
| 58796.341  | is converted to | .58796E 05  |
| -369.7583  | is converted to | -.36976E 03 |
| 0.006873   | is converted to | .68730E-02  |
| 0.2        | is converted to | .20000E 00  |
| -0.0000054 | is converted to | -.54000E-05 |

### Input

Each external value is of field width w with d characters in the fractional part of the value. The value is right-justified with all blanks counting as zeros. A minus sign may precede the value of the exponent. A decimal point placed in the fractional part takes precedence over the d specification. The character E may be present to separate the value and the exponent.



## I/O statements

For a field specification of E10.3:

|            |                 |           |
|------------|-----------------|-----------|
| 123E3      | is converted to | 123.0     |
| 12874E2    | is converted to | 1287.4    |
| -563E-02   | is converted to | -0.00563  |
| 398E00     | is converted to | 0.398     |
| 53876+1    | is converted to | 538.76    |
| 5455-01    | is converted to | 0.5455    |
| -6.7563E05 | is converted to | -675630.0 |

## D CONVERSION

The *D* conversion is used for the input/output of double-precision numbers. It is used exactly as the *E* conversion except the letter *E* is replaced by *D*.

## I CONVERSION

Form

rlw

Only integer data may be processed by this form of conversion.

### Output

Internal values are converted to integer constants. Negative values are preceded by a minus sign. Each field is right-justified and filled with leading blanks.

For a field specification of I6:

|       |                 |       |
|-------|-----------------|-------|
| 281   | is converted to | 281   |
| -3567 | is converted to | -3567 |

If the data require more character positions than allowed by the width, *w*, only the most significant *w* positions are output.

For a field specification of I3:

|       |                 |     |
|-------|-----------------|-----|
| 281   | is converted to | 3*  |
| -6374 | is converted to | -6* |

### Input

External input values are right-justified with the width, *w*. Blanks are counted as zeros. Input values must be integer values. A preceding minus sign may be placed on a value.

For a field specification of I4:

|      |                 |      |
|------|-----------------|------|
| 120  | is converted to | 120  |
| -144 | is converted to | -144 |
| 1 2  | is converted to | 102  |
| -3   | is converted to | -3   |

## A CONVERSION

An *A* format conversion is used in conjunction with a *READ* or *WRITE* statement for the input/output of alphanumeric information to or from a **REAL**, **INTEGER**, or **LOGICAL** list element. The general form is *rAw*, where *r* and *w* are unsigned integer constants. If *r* is one, it can be omitted.

On input, *rAw* will be interpreted to mean that the next *r* successive fields of *w* characters are each to be stored in the associated *REAL* list elements. If *w* is greater than *g*, where *g* is the number of characters a single list element can contain, only the *g* right-most characters will be significant. If *w* is *g* or less, the characters will be left-justified, and the word(s) filled with blanks, if necessary.

On output, *rAw* will be interpreted to mean that the next *r* successive fields of *w* characters are each to be the result of alphanumeric transmission from the specified list elements. If *w* exceeds *g*, only *g* characters of output will be transmitted, preceded by *w - g* blanks. If *w* is *g* or less, the *w* left-most characters of the specified storage element will be transmitted.

## H CONVERSION

In **FORTRAN**, Hollerith information consists of the legal **FORTRAN** character set plus the additional characters

" # : ; ! % & ' |

Information input from the typewriter or paper tape is converted to an internal code used by **FORTRAN**. When this information is output, the internal codes are converted to the appropriate typewriter or paper tape codes.

Form: nHs.

## Output

The number of characters, *n*, in the string, *s*, should contain exactly the number of characters specified so that characters from other fields are not taken as part of the string.

Blanks are counted as characters in the string.

**I/O statements**

**Examples**

| Specification    | External Output |
|------------------|-----------------|
| 1HR              | R               |
| 8HbSTRINGb       | bSTRINGb        |
| 11HX(1,3) = 12.0 | X(1,3) = 12.0   |

**Input**

The w characters in the string, s, are replaced by the next w characters from the input record. The result is a new string in the field specification.

**Example**

| Specification | Input String | Resultant Specification |
|---------------|--------------|-------------------------|
| 5H12345       | ABCDE        | 5HABCDE                 |
| 7HbTRUEbb     | FALSEbb      | 7HFALSEbb               |
| 8Hbbbbbbbbb   | MATRIXbb     | 8HMATRIXbb              |

This feature can be used to change titles, dates, headings, etc., that are output with the program data.

**X SPECIFICATION**

Form: nX.

This specification causes no conversion. On output, n blanks are inserted in the external record. On input, n spaces are skipped from the input record.

**Output Example**

| Specification | Output  |
|---------------|---------|
| 1HA, 4X, 2HBC | AbbbbBC |
| 4X, 3HABC     | bbbbABC |
| 1X, 3HABC, 3X | bABCbbb |

**Input Example**

| Specification  | Input String | Resultant Input |
|----------------|--------------|-----------------|
| F4.1, 3X, F3.0 | 12.5RRR120   | 12.5,120.       |

The RRR characters are ignored by the 3X specification.

## L FORMAT CODE

General Form

`rLw`

where:

`r` is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

`w` is an unsigned integer constant that specifies the number of characters of data.

Logical variables may be read or written by means of the format code `Lw`.

On input, the first T or F encountered in the next `w` characters of the input record causes a value of `.TRUE.` or `.FALSE.`, respectively, to be assigned to the corresponding logical variable. If field `w` consists entirely of blanks, a value of `.FALSE.` is assumed.

On output, a T or F is inserted in the output record corresponding to the value of the logical variable in the I/O list. The single character is preceded by `w - 1` blanks.

## G FORMAT CODE

General Form

`rGw.d`

where:

`r` is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

`w` is an unsigned integer constant specifying the total field length.

`d` is an unsigned integer constant specifying the number of significant digits.

The G format code is a generalized code in that it automatically selects an output format appropriate to the magnitude of the real data.

Input processing is the same as for the `F` conversion.

**I/O statements**

The w portion of the G format code reserves the four right-most positions for a decimal exponent field.

If the real data, n, are in the range  $0.1 \leq n < 10^{**d}$ , where d is the d portion of the format code Gw.d, then this exponent field is blank. Otherwise, the real data are transferred with an E or D decimal exponent depending on the type of the real data.

For the purpose of simplification, the following examples deal with the printed line. However, the concepts apply to all input/output media.

**Example 1**

Assume that the variables A, B, C, and D are of type real whose values are 292.7041, 82.43441, 136.7632, 0.8081945, respectively.

```
1  FORMAT      (G12.4,G12.5,G12.4,G12.7)
2  FORMAT      (G13.4,G13.5,G13.4)
3  FORMAT      (G13.4)
   .
   .
   .
WRITE      (0, n) A, B, C, D
   .
   .
   .
```

Explanation:

- a. If n has been specified as 1, the printed output would be as follows (b represents a blank):

```
Print Position 1                                Print Position 48
↑                                                    ↑
bbb292.7bbbbbb82.434bbbbbb136.8bbbb.8081945bbbb
```

- b. If n has been specified as 2, the printed output would be:

```
Print Position 1                                Print Position 39
↑                                                    ↑
bbbb292.7bbbbbb82.434bbbbbb136.8bbb                                     Line 1
bbbb.8082bbb                                                                    Line 2
```

From the above example, it can be seen that by increasing the field width reserved (w), blanks are inserted.

- c. If  $n$  has been specified as 3, the printed output would be:

Print Position 1

|               |        |
|---------------|--------|
| bbbb292.7bbbb | Line 1 |
| bbbb82.43bbbb | Line 2 |
| bbbb136.8bbbb | Line 3 |
| bbbb.8082bbbb | Line 4 |

From the above example, it can be seen that the same format code is used for each variable in the list. Each repetition of the same format code causes a new line to be printed.

## SCALE FACTOR P

The representation of the data, internally or externally, can be modified by the use of a scale factor followed by the letter P preceding the F, E, G, and D format codes.

The scale factor affects the appropriate conversions in the following manner:

- a. For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

externally represented number equals internally represented number times the quantity ten raised to the  $n$ th power.

- b. For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.
- c. For E and D output, the basic real constant part of the quantity is multiplied by ten to the  $n$ th power and the exponent is reduced by the scale factor.
- d. For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

For example, if input data are in the form  $xx.xxxx$  and it is desired to use this internally in the form  $.xxxxxx$ ; the format code used to effect this change is 2PF7.4.

## Input

As another example, consider the following input data:

27bbb-93.2094bb-175.8041bbbb55.3647

## I/O statements

where b represents a blank.

The following statements:

```
5      FORMAT      (I2,3F11.4)
      .
      .
      .
      READ          (0,5) K,A,B,C
```

cause the variables in the list to assume the following values:

```
K : 27          B : -175.8041
A : -93.2094    C : 55.3647
```

The following statements:

```
5      FORMAT      (I2,1P3F11.4)
      .
      .
      .
      READ          (0,5) K,A,B,C
```

cause the variables in the list to assume the following values:

```
K : 27          B : -17.58041
A : -9.32094    C : 5.53647
```

The following statements:

```
5      FORMAT      (I2,-1P3F11.4)
      .
      .
      .
      READ          (0,5) K,A,B,C
```

causes the variables in the list to assume the following values:

```
K : 27          B : -1758.041
A : -932.094    C : 553.647
```

## Output

Assume the variables K,A,B, and C have the following values:

```
K : 27          B : -175.8041
A : -93.2094    C : 55.3647
```

then the following statements:

```

5      FORMAT      (I2,1P3F11.4)
      .
      .
      .
      WRITE      (0,5) K,A,B,C

```

cause the variables in the list to output the following values:

```

K : 27          B : -1758.041
A : -932.094    C : 553.647

```

The following statements:

```

5      FORMAT      (I2,-1P3F11.4)
      .
      .
      .
      WRITE      (0,5) K,A,B,C

```

cause the variables in the list to output the following values:

```

K : 27          B : -17.5804
A : -9.3209     C : 5.5365

```

For output, when scale factors are used, they have effect only on real data. However, this real data may contain an E or D decimal exponent. A positive scale factor used with real data that contains an E or D decimal exponent increases the number and decreases the exponent. Thus, if the real data were in a form using an E decimal exponent and the statement **FORMAT (1X,I2,3E13.3)** used with an appropriate **WRITE** statement resulted in the following printed line:

```
b27bbbb-.932Eb02bbbb-.175Eb03bbbb.553Eb02
```

the statement **FORMAT (1X,I2,1P3E13.3)** used with the same **WRITE** statement results in the following printed output:

```
b27bbb-9.321Eb01bbb-1.758Eb02bbbb5.536Eb01
```

The statement **FORMAT (1X,I2,-1P3E13.3)** used with the same **WRITE** statement results in the following printed output:

```
27bbbb-.093Eb03bbbb-.018Eb04bbbb.055Eb03
```



## I/O statements

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all format codes following the scale factor within the same *FORMAT* statement. This also applies to format codes enclosed within an additional pair of parentheses.

### / SPECIFICATION

Form

/

Each slash (/) specified in the format causes the termination of a record and processing of the next record. Successive slashes (///...//) cause successive records to be ignored on input, and successive blank records to be written on output. A slash separating two field specifications removes the need for a comma separator. For example,

F5.4/4F10.3.

#### Output Example

For a specification (1HA/1HB/1HC/1HD) the resultant output records are:

A  
B  
C  
D

#### Input Example

Using the four records output from the previous example, an input specification (1H1/1H2//1H3) produces the resultant specification (1HA/1HB//1HD).

## REPEAT SPECIFICATIONS

The A, D, F, E, I, L, and G field specifications can be repeated by using the repeat count *r* in the forms *rAw*, *rDw*, *rFw.d*, *rEw.d*, *rlw*, *rLw*, and *rGw.d*.

#### Examples

4F10.5,F3.6 is equivalent to F10.5,F10.5,F10.5,F10.5,F3.6

2F4.1,2E7.1 is equivalent to F4.1,F4.1,E7.1,E7.1

2F5.2,3I6,2E8.2 is equivalent to F5.2,F5.2,I6,I6,I6,E8.2,E8.2

Repetition of a group of field specifications is accomplished by enclosing the group in parentheses preceded by an integer repeat count. If no repeat count is specified, the count is taken as one.

**Examples**

2(F10.5,I6) is equivalent to F10.5,I6,F10.5,I6

2(E9.3,F7.1/I4) is equivalent to E9.3,F7.1/I4,E9.3,F7.1/I4

3(4F5.0,2E8.2) is equivalent to 4F5.0,2E8.2,4F5.0,2E8.2,  
4F5.0,2E8.2

**Example**

```
50 FORMAT (4X,2(I5,6F8.2)//)
```

The use of additional parentheses (up to two levels) within a *FORMAT* statement is permitted to enable the user to repeat the same format code when transmitting data. For example, the statement:

```
10 FORMAT (2(G10.6,G7.1),G4)
```

is equivalent to

```
10 FORMAT (G10.6,G7.1,G10.6,G7.1,G4)
```

If the data exists with a D decimal exponent, it is transferred with this D decimal exponent.

If a multiline listing is desired such that the first two lines are to be printed according to a special format and all remaining lines according to another format, the last format code in the statement should be enclosed in a second pair of parentheses. For example, in the statement:

```
FORMAT(G2,2G3.1/G10.8/(3G5.1))
```

If more data items are to be transmitted after the format codes have been completely used, the format repeats from the last left parenthesis. Thus, the printed output would take the following form:

```
G2,G3.1,G3.1
  G10.8
G5.1,G5.1,G5.1
G5.1,G5.1,G5.1
  .
  .
  .
```

## I/O statements

As another example, consider the following statement:

```
FORMAT(G2/2(G3,G6.1),G9.7)
```

If 13 data items are to be transmitted, the printed output on a *WRITE* statement takes the following form:

```
      G2  
    G3,G6.1,G3,G6.1,G9.7  
    G3,G6.1,G3,G6.1,G9.7  
      G3,G6.1
```

## FORMAT CONTROL AND LINE INTERACTION

Execution of a formatted *READ* or *WRITE* statement initiates format control. The conversion performed on data depends on information jointly provided by the next element of the input-output list and the next field specification of the *FORMAT* statement. If there is a list, at least one field specification of type D, E, F, G, L, A, or I should be present in the *FORMAT* statement.

Execution of a formatted *READ* statement causes one record to be input. Each D, E, F, G, L, A, or I specification has a corresponding element in the list. Each H or X specification has no corresponding element in the list and the format control communicates information directly to the record. When a slash is encountered or the entire input record is processed, the record is terminated. If more input is necessary, the next record is input. Any unprocessed characters of a record are skipped when a slash is encountered.

A *READ* statement is terminated upon ending the list if:

- a. The next specification is A, D, E, F, G, I, or L.
- b. The format control has reached the last outer right parenthesis of the *FORMAT* statement.

If the list ends and the next specification is an H or X, data are processed (with the possibility of additional records being input) until one of the two above conditions is met.

If the format control reaches the right-most parenthesis of the *FORMAT* statement and more list remains to be processed, the following steps are taken:

- a. A new record is input and remaining data in the previous record ignored.
- b. Format control reverts to the point immediately following the last left parenthesis.

If group repeat specifications exist in the format, this point is at the right-most group of the format. The repeat count is not taken into consideration. If no groups are present, the format is started from the beginning.

When a formatted *WRITE* statement is executed, records are written each time 120 characters have been processed, a slash is encountered, or the format control terminates. The format control terminates by one of the two methods described for *READ* termination. Incomplete records are filled with blanks to maintain standard record lengths.



## SECTION 7 — PROGRAMS AND SUBPROGRAMS

### GENERAL

An executable **FORTRAN** program consists of a main program and any required subprograms. Subprograms may be defined by the programmer or contained in the system library. Each program or subprogram must contain at least one executable statement.

### MAIN PROGRAMS

A main program is a program unit consisting of a set of **FORTRAN** statements, comment lines, and an *END* line. The program may be preceded by specification statements.

A main program cannot contain a subprogram definition statement, namely:

- a FUNCTION statement
- a SUBROUTINE statement
- a BLOCK DATA statement

A main program may contain calls to other subprograms or may contain statement function subprograms.

An MOS main program can accept a main program entry name definition of the following format:

NAME N1, N2, ..., Nn

where:

N1, N2, ..., Nn are entry names by which the main program can be referenced

### SUBPROGRAMS

Subprograms are program units which may be called by other programs or subprograms. Subprograms are categorized as one of the following:

- PROCEDURE SUBPROGRAMS
- FUNCTION subprogram
- SUBROUTINE subprogram

## subprograms

### SPECIFICATION subprogram BLOCK DATA subprogram

Functions are programmed procedures that are often used to provide solutions to mathematical functions. Function references may be used in the same manner as references to variables in an expression. For example:  $X = AB * \text{SIN}(Y) \cdot C * \text{COS}(Y * Z)$ , where *SIN* is the name of the sine function, *COS* is the name of the cosine function, and (*Y*) and (*Y\*Z*) are their respective argument lists. The value returned for a function reference is of the same mode as the function name, corresponding to the rules for real and integer symbolic names.

## Function Subprograms

A function subprogram is defined external to the program unit by which it is referenced. A function subprogram is defined by having as its first statement, other than comment lines, a statement of the form:

```
FUNCTION f(a1, a2, a3, ..., an)
```

where

*f* is the symbolic name of the function and

*ai* represent dummy arguments.

Each *ai* is either a variable name, array name, or an external procedure name. The *ai* defines the type, number, and order of the *FUNCTION* arguments. A function subprogram must have at least one argument.

A function subprogram is executed at the first executable statement following the *FUNCTION* statement. Specification statements (*DIMENSION*, *COMMON*, and *EQUIVALENCE*) may immediately follow the *FUNCTION* statement. If present, these must precede any other statement, excluding comments. The symbolic names of the dummy arguments, *ai*, may not appear in an *EQUIVALENCE* or *COMMON* statement.

A function subprogram must contain at least one *RETURN* statement, and the last statement executed in a *FUNCTION* must be a *RETURN* statement. The function subprogram is ended by an *END* line.

The symbolic name, *f*, of the *FUNCTION* must appear as a variable name within the subprogram. The value returned for a *FUNCTION* is the last value assigned to this name prior to execution of a *RETURN* statement. The type of the *FUNCTION* value is as for a variable (page 2-2).

The symbolic name of the function must not appear in any nonexecutable statement within the subprogram.

**Example**

```

      FUNCTION XP(A,B,I)
      DIMENSION B(10)
      XP = 0.
      DO 1 J = 1,10
1      XP = (A*B(J))*I + XP
      RETURN
      END

```

A *FUNCTION* is executed with a function reference by a main program or another subprogram. The actual arguments in the call must correspond in type, number, and order with the *FUNCTION* dummy arguments. If a dummy argument of a *FUNCTION* is an array name, the corresponding actual argument must be an array name.

**Example:**

A call for the example *FUNCTION* shown above would be:  
 $W = XP(R,S,K)$ , where S is an array.

**Type Specification of FUNCTION Subprogram**

In addition to declaring the type of a *FUNCTION* name by the predefined convention, there exists the option of explicitly specifying the type of a *FUNCTION* name within the function statement.

**General Form**

Type *FUNCTION* name (a1, a2, a3, ..., an)

where:

Type is integer, real, double precision, complex, or logical.

name is the name of the *FUNCTION* subprogram.

a1, a2, a3,...,an are nonsubscripted variables, or dummy names of subroutine or other *FUNCTION* subprograms. (There must be at least one argument in the argument list.)



## subprograms

### Example 1

```
REAL FUNCTION SOMEF (A,B)
  .
  .
  .
  SOMEF = A**2 + B**2
  .
  .
  .
  RETURN
  END
```

### Example 2

```
INTEGER FUNCTION CALC (X,Y,Z)
  .
  .
  .
  CALC = X + Y + Z**2
  .
  .
  .
  RETURN
  END
```

Explanation:

The *FUNCTION* subprograms *SOMEF* and *CALC* in Examples 1 and 2 are declared as type *REAL* and *INTEGER*, respectively.

## Subroutine Subprograms

A subroutine subprogram is defined external to the program unit that references it. Subroutines, unlike functions, do not have values associated with them and cannot be referenced in an expression. Subroutines are accessed by *CALL* statements.

A subroutine subprogram is defined by having as its first statement, other than comment lines, a statement of the form: *SUBROUTINE S (a1, a2, a3, ..., an)* or *SUBROUTINE S*, where *S* is the symbolic name of the subroutine and *a<sub>i</sub>* represents dummy arguments of the subroutine. Each *a<sub>i</sub>* is either a variable or an array name, or the name of an external procedure. If no arguments are passed to the subroutine, the second form is used.

The symbolic name of the subroutine must not appear in any statement in the subprogram. The symbolic names of the dummy arguments may not appear in *COMMON* or *EQUIVALENCE* statements.

A subroutine is executed at the first executable statement. Specification statements must immediately follow the *SUBROUTINE* statement and precede any executable statement. A subroutine must have at least one *RETURN* statement. The last statement executed by a subroutine must be a *RETURN* statement.

#### Example

```

SUBROUTINE R(A,I,Z)
  DIMENSION A(10)
  Z = 0
  DO 1 J = 1,10
1   Z = Z + A(J)**I
  RETURN
END

```

A subroutine is referenced with a *CALL* statement. The argument list in the reference must agree in type, number, and order with the dummy arguments of the subroutine. If a dummy argument is an array name, the corresponding actual argument must be an array name.

#### Example:

A call for the example SUBROUTINE above would be: *CALL R(T,K,D)*  
where T is an array.

## Block Data Subprogram

To initialize variables in a *COMMON* block, a separate subprogram must be written. This separate subprogram contains only the *DATA*, *COMMON*, *DIMENSION*, *EQUIVALENCE*, and *TYPE* statements associated with the data being defined. Data may be initialized in labeled (named), but not unlabeled, *COMMON* by the *BLOCK DATA* subprogram.

#### General Form

```

BLOCK DATA
  .
  .
  .
END

```

## subprograms

- a. The *BLOCK DATA* subprogram may not contain any executable statements.
- b. The *BLOCK DATA* statement must be the first statement in the subprogram.
- c. All elements of a *COMMON* block must be listed in the *COMMON* statement, even though they are not all initialized; for example, the variable *A* in the *COMMON* statement in the following example does not appear in the data initialization statement.

```
BLOCK DATA
COMPLEX C
COMMON/ELN/C,A,B/RMG/Z,Y
DATA C/(2.4.3.769)/
```

- d. Data may be entered into more than one *COMMON* block in a single *BLOCK DATA* subprogram.

## Data Initialization Statement

### General Form

```
DATA k1,...,kn/j1*d1,...,jn*dn/,kn + 1,...,k/jn + 1*dn + 1,...,j*d/,...
```

where:

$k_1, \dots, k$  are variables and/or subscripted variables (in which case the subscripts must be integer constants), or array names, or implied DO lists

$d_1, \dots, d$  are values representing integer, real, double-precision, complex, logical or Hollerith data constants.

$j_1^*, \dots, j^*$  represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of  $d_1, \dots, d_n$ .

A data initialization statement defines initial values of variables and array elements. There must be a one-to-one correspondence between these variables (i.e.,  $k_1, \dots, k$ ) and the data constants (i.e.,  $d_1, \dots, d$ ).

### Example 1

```
DIMENSION D(10)
```

```
DATA A,B,C/5.0,6.1,7.3/,D(1),D(2),D(3),D(4),D(5)/5*1.0/
```

Explanation:

The **DATA** statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3, respectively. In addition, the statement specifies that the first five variables in array D are to be initialized to 1.0.

### Example 2

```
DIMENSION A(5),B(3),L(2)
```

```
DATA A(1),A(2),A(3),A(4),A(5)/5*1.0/,B(1),B(2)/2*5.0/,L(1),L(2)/.TRUE.,.FALSE./
```

Explanation:

The **DATA** statement specifies that all the variables in array A are to be initialized to 1.0 and the first two elements of array B are to be initialized to 5.0. The logical variables, (L(1) and L(2)), in array L are initialized to .TRUE. and .FALSE., respectively.

An initially defined variable, or any element, may not be in blank common. However, in a labeled common block, they may be initially defined only in a block data subprogram.

### Example 3

```
DIMENSION A(3), B(3,2)
```

```
DATA A/1.0,2.0,3.0/,((B(I,J),J = 1,2),I = 1,3)/6*5./
```

Explanation:

The **DATA** statement loads real numbers 1.0, 2.0, and 3.0 into array A. It also loads real number 5. into every element of array B.

## STATEMENT FUNCTIONS

A statement function is defined internal to the program unit in which it is referenced. All statement functions must precede the first executable statement and must follow any specification statements of the program unit.

A statement function is defined in a single expression of the form:  $f(a_1, a_2, a_3, \dots, a_n) = e$ , where  $f$  is the function name,  $a_i$  represents arguments, and  $e$  is an expression. The resultant value of the function is either a real or integer value corresponding to the function name. The  $a_i$  are distinct variable names and are called dummy arguments. They serve to indicate the type, number, and order of the function arguments. The expression  $e$  is an arithmetic expression and may contain references to previously defined statement functions.

## subprograms

A statement function is referenced by a function call,  $f(a_1, a_2, a_3, \dots, a_n)$ , appearing in an arithmetic expression. A statement function may be referenced only within the program unit in which it is defined. The arguments used in the reference must agree in type, number, and order with the corresponding dummy arguments.

### Example

The statement function:

$$SF(X) = A * X ** 2 + B * X + C$$

can be referenced in the program by:

$$W = SF(Y)$$

## INTRINSIC FUNCTIONS

Intrinsic functions are commonly used subprograms contained in the **FORTRAN** library. The symbolic names and meanings of the intrinsic functions are shown in Figure 7-1.

An intrinsic function is referenced by a function call in an arithmetic expression. The arguments in the argument list must agree in type, number, and order with those shown in Figure 7-1.

### Example

```
IF (SIGN(W,X)) 1,2,2
1  W = ABS(X) - ABS(Y)
2  S = W*FLOAT(I*J)
   K = IFIX(X) + J
```

## BASIC EXTERNAL FUNCTIONS

Basic external *FUNCTIONS* are standard subprograms contained in the **FORTRAN** library. These are referenced in the same manner as normal *FUNCTIONS*. The symbolic names and meanings of the basic external *FUNCTIONS* are shown in Figure 7-2.

## DUMMY ARGUMENTS

Dummy arguments provide a means of passing information between a subprogram and the program or subprogram that called it. Both function and subroutine subprograms may have dummy arguments. A subroutine need not have any, while a function must have at least one. Dummies provide definitions of the data type, number, and sequence of subprogram parameters.

A dummy can be classified within a subprogram as a variable, an array, or an external procedure name. The actual arguments defined by a calling program or subprogram to which a dummy can correspond are: Hollerith constants, variables, array elements, arrays, expressions, and external procedure names.

Within a subprogram, a dummy can be used in much the same way as any other variable or array. A dummy can not appear in a *COMMON* or *EQUIVALENCE* statement.

The actual arguments (except for Hollerith constants) used in a calling statement agree in data type with the corresponding dummy arguments, that is, real to real, integer to integer, and array to array. If an actual argument is an expression, the result of the expression should correspond in data type to the dummy.

A dummy array is defined as an argument which appears in a *DIMENSION* statement in the subprogram. A dummy array does not occupy any storage but tells the subprogram that the argument supplied in the calling statement defines the first element of an actual array. The calling argument need not have the same dimensions as the dummy array. Useful operations can sometimes be performed by defining different dimensions for the dummy and calling arguments.

#### Example

|            |            |
|------------|------------|
| DIMENSION  | A(10,10)   |
| CALL       | FM(A(6,1)) |
| .          |            |
| .          |            |
| .          |            |
| SUBROUTINE | FM(B)      |
| DIMENSION  | B(50)      |

For this case, one-dimensional dummy array B corresponds to the last half of two-dimensional array A. If the calling statement were *CALL FM (A)*, dummy array B would correspond to the first half of array A.

## ADJUSTABLE DIMENSIONS

As shown in the previous examples, the maximum value of each subscript in an array is specified by a numeric value. These numeric values (maximum value of each subscript) are known as the absolute dimensions of an array and may never be changed. However, if an array is used in a subprogram (page 7-1) and is not in *COMMON*, the size of this array does not have to be explicitly declared in the subprogram by a numeric value. That is, the specification statement, appearing in a subprogram, may contain integer variables that specify the size of the array. These integer variables must be either actual or implicit subprogram arguments. When the subprogram is called, these integer variables receive their values from the calling program. Thus, the dimensions (size) of a dummy array appearing in a subprogram are adjustable and may change each time the subprogram is called. Integer variables that provide dimension information may not be redefined within the subprogram.

## subprograms

The absolute dimensions of an array must be declared in a calling program. The adjustable dimensions of an array, appearing in a subprogram, should be less than or equal to the absolute dimensions of that array as declared in the calling program.

The following example illustrates the use of adjustable dimensions.

### Example

#### CALLING PROGRAM

```
DIMENSION A(5,5)
.
.
.
CALL MAPMY(...,A,2,3,...)
.
.
.
```

#### SUBPROGRAM

```
SUBROUTINE MAPMY
(...,R,L,M,...)
.
.
.
DIMENSION...,R(L,M),...
.
.
.
DO 100 I = 1,L
.
.
.
```

#### Explanation:

The statement `DIMENSION A(5,5)` appearing in the calling program declares the absolute dimensions of array A. When subroutine MAPMY is called, dummy argument R assumes array name A and dummy arguments L and M assume the values 2 and 3, respectively. The correspondence between the subscripted variables of arrays A and R is shown in the following example.

`R(1,1)R(2,1)R(1,2)R(2,2)R(1,3)R(2,3)`

`A(1,1)A(2,1)A(3,1)A(4,1)A(5,1)A(1,2)A(2,2)...`

Thus, in the calling program the subscripted variable `A(1,2)` refers to the sixth subscripted variable in array A. However, in subprogram MAPMY, the subscripted variable `R(1,2)` refers to the third subscripted variable in array A, namely, `A(3,1)`. This is so because the dimensions of array R as declared in the subprogram are not the same as those in the calling program.

If the absolute dimensions in the calling program were the same as the adjusted dimensions in the subprogram, the subscripted variables `R(1,1)` through `R(5,5)` in the subprogram would always refer to the same storage locations as specified by the subscripted variables `A(1,1)` through `A(5,5)` in the calling program, respectively.

The numbers 2 and 3, which became the adjusted dimension of dummy array R, could also have been variables in the argument list or implicit arguments in a *COMMON* block. For example, assume that the following statement appeared in the calling program.

```
CALL MAPMY (...A,I,J,...)
```

Then as long as the values of I and J are previously defined, the arguments may be variables. In addition, the variable dimension size may be passed through more than one subprogram level. For example, the subprogram MAPMY could have contained a call statement to another subprogram in which dimension information about A could have been passed.

Dummy variables (e.g., L and M) may be used as dimensions of an array only in a *FUNCTION* or *SUBROUTINE* subprogram.

## EXTERNAL STATEMENT

General Form

```
EXTERNAL a,b,c,...
```

where:

a,b,c,... are names of subprograms that are passed as arguments to other subprograms.

The name of any subprogram passed as an argument to another subprogram must appear in an *EXTERNAL* statement in the calling program. For example, assume that SUB and MULT are subprogram names in the following statements:

### Example

| CALLING PROGRAM    |   | SUBPROGRAM            |
|--------------------|---|-----------------------|
| EXTERNAL MULT      |   | SUBROUTINE SUB(K,Y,Z) |
| .                  |   | IF(K)4,6,6            |
| .                  | 4 | D = Y(K,Z**2)         |
| .                  |   | .                     |
| CALL SUB(J,MULT,C) |   | .                     |
| .                  |   | .                     |
| .                  | 6 | RETURN                |
| .                  |   | END                   |

Explanation:

In this example, the subprogram name MULT is used as an argument in subprogram SUB. The subprogram name MULT is passed to dummy variable Y, and variables J and C are passed to dummy variables K and Z, respectively. Subprogram MULT is called and executed only if the value of K is negative.



| Intrinsic Function      | Definition                                   | Arguments | Name                                             | Type of Argument                                       | Type of Function                                       |
|-------------------------|----------------------------------------------|-----------|--------------------------------------------------|--------------------------------------------------------|--------------------------------------------------------|
| Absolute Value          | $ a $                                        | 1         | ABS<br>IABS<br>DABS                              | Real<br>Integer<br>Double                              | Real<br>Integer<br>Double                              |
| Truncation              | Sign of $a$ times largest integer $\leq  a $ | 1         | AIN<br>INT<br>IDINT                              | Real<br>Real<br>Double                                 | Real<br>Integer<br>Integer                             |
| Remaindering*           | $a_1 \pmod{a_2}$                             | 2         | AMOD<br>MOD                                      | Real<br>Integer                                        | Real<br>Integer                                        |
| Choosing Largest Value  | $\text{Max}(a_1, a_2, \dots)$                | $\geq 2$  | AMAX0<br>AMAX1<br>MAX0<br>MAX1                   | Integer<br>Real<br>Integer<br>Real                     | Real<br>Real<br>Integer<br>Integer                     |
| Choosing Smallest Value | $\text{Min}(a_1, a_2, \dots)$                | $\geq 2$  | DMAX1<br>AMIN0<br>AMIN1<br>MIN0<br>MIN1<br>DMIN1 | Double<br>Integer<br>Real<br>Integer<br>Real<br>Double | Double<br>Real<br>Real<br>Integer<br>Integer<br>Double |
| Float                   | Conversion from integer to real              | 1         | FLOAT                                            | Integer                                                | Real                                                   |
| Fix                     | Conversion from real to integer.             | 1         | FIX                                              | Real                                                   | Integer                                                |
| Transfer of Sign        | Sign of $a_2$ times $ a_1 $                  | 2         | SIGN<br>ISIGN<br>DSIGN                           | Real<br>Integer<br>Double                              | Real<br>Integer<br>Double                              |

Figure 7-1. Intrinsic Functions

| Intrinsic Function                                         | Definition             | Arguments | Name        | Type of Argument | Type of Function |
|------------------------------------------------------------|------------------------|-----------|-------------|------------------|------------------|
| Positive Difference                                        | $a_1 - \min(a_1, a_2)$ | 2         | DIM<br>IDIM | Real<br>Integer  | Real<br>Integer  |
| Obtain Most Significant Part of Double Precision Argument  |                        | 1         | SNGL        | Double           | Real             |
| Obtain Real Part of Complex Argument                       |                        | 1         | REAL        | Complex          | Real             |
| Obtain Imaginary Part of Complex Argument                  |                        | 1         | AIMAG       | Complex          | Real             |
| Express Single Precision Argument in Double Precision Form |                        | 1         | DBLE        | Real             | Double           |
| Express Two Real Arguments in Complex Form                 | $a_1 + a_2\sqrt{-1}$   | 2         | CMPLX       | Real             | Complex          |
| Obtain Conjugate of a Complex Argument                     |                        | 1         | CONJG       | Complex          | Complex          |

\*The function MOD or AMOD ( $a_1, a_2$ ) is defined as  $a_1 - [a_1/a_2] a_2$ , where  $[x]$  is the integer whose magnitude does not exceed the magnitude of  $x$  and whose sign is the same as  $x$ .

Figure 7-1. Intrinsic Functions (2 of 2)

| External Functions   | Definition      | Arguments | Name   | Type of Argument | Type of Function |
|----------------------|-----------------|-----------|--------|------------------|------------------|
| Exponential          | $e^a$           | 1         | EXP    | Real             | Real             |
|                      |                 |           | DEXP   | Double           | Double           |
|                      |                 |           | CEXP   | Complex          | Complex          |
| Natural Logarithm    | $\log_e (a)$    | 1         | ALOG   | Real             | Real             |
|                      |                 |           | DLOG   | Double           | Double           |
|                      |                 |           | CLOG   | Complex          | Complex          |
| Common Logarithm     | $\log_{10} (a)$ | 1         | ALOG10 | Real             | Real             |
|                      |                 |           | DLOG10 | Double           | Double           |
| Trigonometric Sine   | $\sin(a)$       | 1         | SIN    | Real             | Real             |
|                      |                 |           | DSIN   | Double           | Double           |
|                      |                 |           | CSIN   | Complex          | Complex          |
| Trigonometric Cosine | $\cos(a)$       | 1         | COS    | Real             | Real             |
|                      |                 |           | DCOS   | Double           | Double           |
|                      |                 |           | CCOS   | Complex          | Complex          |
| Hyperbolic Tangent   | $\tanh(a)$      | 1         | TANH   | Real             | Real             |
| Square Root          | $(a)^{1/2}$     | 1         | SQRT   | Real             | Real             |
|                      |                 |           | DSQRT  | Double           | Double           |
|                      |                 |           | CSQRT  | Complex          | Complex          |

Figure 7-2. Basic External Functions (1 of 2)

| External Functions | Definition         | Arguments | Name   | Type of Argument | Type of Function |
|--------------------|--------------------|-----------|--------|------------------|------------------|
| Arctangent         | $\arctan(a)$       | 1         | ATAN   | Real             | Real             |
|                    |                    |           | DATAN  | Double           | Double           |
|                    | $\arctan(a_1/a_2)$ | 2         | ATAN2  | Real             | Real             |
|                    |                    |           | DATAN2 | Double           | Double           |
| Remaindering*      | $a_1 \pmod{a_2}$   | 2         | DMOD   | Double           | Double           |
| Modulus            |                    | 1         | CABS   | Complex          | Real             |

\*The function DMOD ( $a_1, a_2$ ) is defined as  $a_1 - [a_1/a_2] a_2$ , where  $[x]$  is the integer whose magnitude does not exceed the magnitude of  $x$  and whose sign is the same as the sign of  $x$ .

Figure 7-2. Basic External Functions (2 of 2)



## SECTION 8 — OPERATING INSTRUCTIONS

### GENERAL

The Varian Data Machines (ANSI) FORTRAN IV Compiler is available as a stand-alone program or as another processor within the Varian 620 Master Operating System (MOS). Whereas the **FORTRAN IV** stand-alone program operates within a minimum configuration of 8,192 words of memory and a 33/35 ASR Teletype, the minimum for MOS FORTRAN IV is 12,288 words of memory plus the required MOS peripheral devices.

**FORTRAN** programs and subprograms are compiled by the FORTRAN IV compiler. The FORTRAN IV loader loads main programs and all required subprograms into memory for execution. The FORTRAN IV run-time library provides input/output, control, and mathematical functions required at execution time.

### COMPILER OPERATING INSTRUCTIONS

The FORTRAN IV compiler translates **FORTRAN IV** source programs to relocatable machine language programs in a single pass. Error diagnostics, source listings and object listings are provided. Input/output and listing options are selected for each program to be compiled. The memory reserved for integer and logical items can be selected as one or two words per item.

### Stand-Alone FORTRAN IV

The Varian stand-alone FORTRAN IV compiler is supplied as an absolute binary object tape. The compiler is loaded into memory by the standard binary loader and occupies approximately 016650 words of memory. (See the reference handbook for the procedure for loading absolute object programs.)

The compiler is entered at location 0. Upon entry, the compiler executes a HALT with the P register set to 4, and the A register set to the upper limit of compiler-used memory (015777 standard for 8K). This limit can be modified by resetting the A register. To compile the programs, press RUN.

### MOS FORTRAN IV

The initiation of the MOS FORTRAN IV compiler is accomplished by entering the control directive

/FORTRAN (or /F) P1,P2,...,Pn.

This control directive directs the Executive program to call the System Loader to load the FORTRAN IV compiler and commence compilation. The parameter string specifies optional tasks that are to be performed. These options are:

- B No binary object program output desired
- D Integer and logical items are assigned two words.
- L Load and go operation desired.
- M No memory map desired.
- N No source listing desired.
- O Octal listing of object program desired.
- X Conditional compilation desired. (Source records with an X in column 1 will be compiled.)

Input/output assignments during compilation are made through the /ASSIGN control directive (see 620 Master Operating System Reference Manual). The FORTRAN IV compiler uses the following logical units:

|               |               |
|---------------|---------------|
| Source input  | PI            |
| Object output | BO            |
| Listing       | LO            |
| Load and go   | GO (optional) |

## LOADER OPERATING INSTRUCTIONS

### Stand-Alone FORTRAN IV

The loader loads relocatable object programs produced by the FORTRAN IV compiler, and FORTRAN-compatible subprograms produced by the DAS 8A assembler. Object program input is from either paper or magnetic tape selected at the Teletype keyboard. Maps and error diagnostics appear on the Teletype printer.

The loader is on an absolute binary object tape and is loaded into memory by the binary loader. The FORTRAN loader occupies locations 000 through 007 and 0x4000 through 0x5773, where x is 0, 1, 2, 3, 4, 5, 6, or 7, depending on whether the computer memory size is 4K, 8K, 12K, 16K, 20K, 24K, 28K, or 32K words. The loader can be started from location 0x4146 or location 0. (Locations 0200 through 0377 are reserved for loader-generated pointers). The first program to be loaded must be a FORTRAN-compiled main program. Prepare the input unit, clear the registers, and press RUN. A message then appears on the Teletype, requesting input selection.

For high-speed paper tape reader input, type P; for Teletype paper tape reader input, type T. For magnetic tape input, type the unit number (0, 1, 2, or 3). If the selected unit is attached and ready, the loader loads the main program at 0500. The Teletype then types RQ, followed by a list of the required subroutines, followed by another input selection request. At this time, load all required subprograms.

The FORTRAN loader nominally occupies the area just below and adjacent to the AID program. Setting SENSE switch 1 at load time, however, causes the FORTRAN loader to relocate and overlay the AID program, thus occupying the area below and adjacent to the BLD II program. With this option, the FORTRAN loader occupies locations 000 through 007 and 0x5400 through 0x7373, and is entered at location 0x5540.

For most efficient use of the loader, load standard FORTRAN library subprograms in the following order:

- a. Run-time input/output
- b. Complex math functions
- c. Double-precision math functions
- d. Single-precision math functions
- e. Double-precision math library
- f. Single-precision math library
- g. Run-time utility

Prepare and select the input unit as for main programs. The loader loads all required subprograms until an end-of-tape record or an end of file is detected, at which time the list of required subprograms is generated and input selection is again requested. Note that the end-of-tape record is not produced for subprograms by either the compiler or the assembler, but is supplied as a separate tape labeled FORTRAN end-of-tape to be spliced to the end of the user's subprogram library tapes. Standard FORTRAN library tapes have end-of-tape records.

If two or more subprograms have the same name, only the first such subprogram input is loaded. When all required subprograms have been loaded, GO is typed followed by the load map, which lists each loaded subprogram with its entry point. To execute the main program, press RUN. Execution of the main program can be forced by running at location 0500.

An error in loading outputs an error message and the loading map. A minus sign precedes the address of each subprogram that has not been loaded. In this case, the address is the last location where the subprogram is requested. All errors except checksum are nonrecoverable. Correct the error and reinitialize the loading process. Loading errors are listed below.



## operation

| Error | Comment                                                                              |
|-------|--------------------------------------------------------------------------------------|
| CK    | Checksum. Backspace the input tape one record and press RUN for another attempt.     |
| AR    | Area reference. An attempt has been made to load a value to an area not yet defined. |
| CE    | Compiler error. A terminating error occurred at compilation time.                    |
| CS    | Common size. A second use of blank common is larger than the initially defined area. |

| Error | Comment                                                          |
|-------|------------------------------------------------------------------|
| SZ    | Program size. The program is too large for the available memory. |

All FORTRAN main programs are loaded and entered at location 0500. Required subprograms are loaded as they are input to successive blocks of memory. Common storage overlays the FORTRAN loader. AID II routines and the absolute binary loader are in memory at their standard locations. Locations 0 through 077 are unused and locations 0200 through 0377 contain program and data pointers used by the program and subprograms to be executed.

To execute a FORTRAN program, initialize the input/output devices selected, clear the console registers, set the program counter to 0500, press SYSTEM RESET, and RUN.

There are three types of programmed halts: STOP, PAUSE, and EXIT. STOP causes a HALT 0777 with the stop number in the A register and -1 in the B register. STOP implies an end of job. PAUSE causes a HALT 0000 with the pause number in the A register, and zero in the B register. The program can be continued by pressing SYSTEM RESET and RUN. EXIT causes a HALT 0777 with -1 in the A and B registers, and signifies end of job. All programmed halts display error bits in the X register, as follows:

| Bit | Indication              |
|-----|-------------------------|
| 0   | Floating-point overflow |
| 1   | Division check error    |
| 2   | Fixed-point overflow    |
| 3   | Indeterminate function  |
| 4   | Log error               |
| 5   | Square root error       |
| 6   | GO TO error             |

The following error halts are generated by the run-time I/O package. These errors give a four-character message on the Teletype, followed by a call to EXIT.

| Message | Definition                                      |
|---------|-------------------------------------------------|
| FRMT    | Format error                                    |
| MODE    | Data mode error<br>(floating point vs. integer) |
| DATA    | Input data field error                          |
| UNIT    | Unit not attached or not available              |

| Message | Definition                    |
|---------|-------------------------------|
| TAPE    | Checksum or tape parity error |

All binary input/output records are a fixed length of 64 words. Paper tape records are punched with a record mark and checksum. Checksum errors encountered on input cause TAPE errors. Reading a file mark from magnetic tape causes a tape error halt.

BCD records can be fixed or variable length, depending on the device. However, only the first 80 characters are processed.

Keyboard and paper tape records are variable in length and are terminated by a carriage return followed by a line feed. The character - can be used to clear the input buffer and reset to column 1. Illegal characters are ignored. For keyboard input, the Teletype bell signals that BCD input is required. Maximum length of records is 72 characters for keyboard and 80 characters for paper tape. Card records are a fixed length of 80 characters. Illegal characters are treated as blanks.

Magnetic tape records are a fixed length of 84 characters, and should be card images with blank padding characters. Illegal characters are treated as blanks.

Note that the model 33 Teletype paper tape punch must be turned on and off by the operator.

Line printer records are 120 characters. Printing is left-justified with unused positions set to blanks.

## MOS FORTRAN IV

To run a program compiled under MOS FORTRAN IV, initialize the MOS, and load the compiled object program with the MOS directive (see Varian 620 Master Operating System Reference Manual).

`/LOAD (or /L)`

## INPUT/OUTPUT DEVICE SPECIFICATIONS

### Stand-Alone FORTRAN IV

For each program to be compiled a ? = will be typed on the teletype printer requesting input/output selection. The operator must respond by typing one of the following characters to indicate the input device.

|     |                       |
|-----|-----------------------|
| C   | Card reader           |
| K   | Teletype keyboard     |
| P   | High speed paper tape |
| T   | TTY paper tape        |
| 0-3 | Magnetic tape unit #  |

followed by one of the following characters to indicate the output device

|     |                       |
|-----|-----------------------|
| C   | Card punch            |
| P   | High speed paper tape |
| T   | TTY paper tape        |
| 0-3 | Magnetic tape unit #  |

followed by an optional listing selection character

|   |                                 |
|---|---------------------------------|
| S | Source listing                  |
| O | Object listing                  |
| B | Both source and object listings |

followed by an integer/logical size allocation indicator

|       |                                    |
|-------|------------------------------------|
| blank | One-word integer and logical items |
| D     | Two-word integer and logical items |

followed by the character >, followed by the one- to six-character program name (optional), followed by @ for a carriage return and line feed. For example

=CPSD > MATRIX @

where

|   |                                                      |
|---|------------------------------------------------------|
| C | indicates card input                                 |
| P | indicates high speed paper tape output               |
| S | indicates source listing of the program named MATRIX |
| D | indicates two-word integer and logical items         |

The integer/logical size allocation indicator is optional. If omitted, only one word will be allocated for each integer and logical item in the program.

Following input/output selection, source statements are read and object records are output through the selected devices. Error diagnostics and selected list options are printed on the teletype or line printer (if available). Upon detection of an END statement followed by a nonblank statement, the compiler produces a program map listing all variables, constants (in octal), and required subprograms. After listing the program map the compiler types a = to permit compilation of another program.

## MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual.

### COMPILER INPUT RECORDS

#### Stand-Alone FORTRAN IV

Input to the compiler is a series of **FORTRAN** statements, each of which appears in one or more input records. Records can be fixed or variable length depending on the device; however, only the first 72 characters of each record are used by the compiler. Any illegal characters are treated as blanks. END statements must be followed by at least one nonblank record (another END statement is suggested).

Keyboard and paper tape records are variable length and are terminated by a carriage return and line feed in that order. The character > can be used to TAB to column 7, and the character - can be used to clear the input buffer and reset to column 1. For keyboard input the teletype bell is used to notify the operator that source input is required. Position the paper tape so that the first character is at the read station.

Card records are a fixed length of 80 characters. Magnetic tape records should be card images.

## MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual.

### COMPILER OUTPUT RECORDS

#### Stand-Alone FORTRAN IV

Object records are output as they are created. Paper tape object programs are punched with leader and trailer records.

Magnetic tape object programs are terminated by an end of file and a backspace to permit the stringing of compiled subprograms. All main programs are terminated by an end-of-tape record.

All error diagnostics are of the form

```
ERR xx a...a
```

where

xx is a number from 1 to 18 (notification error), or T1 to T9 (termination error)

a...a represents the last (up to 12) characters encountered in the statement being processed.

## **operation**

The rightmost character indicates the point where the error was discovered (the character - indicates end of statement). If a termination error is discovered, object output is terminated, but source code is continued to detect any further errors.

## **MOS FORTRAN IV**

See the Varian 620 Master Operating System Reference Manual.

## **NOTIFICATION ERRORS**

### **Stand-Alone FORTRAN IV**

- 1 Construction
- 2 Usage
- 3 Mode
- 4 Illegal *DO* termination
- 5 Improper statement number
- 6 *COMMON* base lowered
- 7 Illegal equivalence group
- 8 Reference to nonexecutable statement
- 9 No path to this statement
- 10 Multiply defined statement number
- 11 Invalid format construction
- 12 Spelling error
- 13 Format with no statement number
- 14 Function not used as variable
- 15 Truncated value
- 16 Statement out of order
- 17 More than 29 *COMMON* regions
- 18 Non*COMMON* data

## **MOS FORTRAN IV**

See the Varian 620 Master Operating System Reference Manual.

## **TERMINATION ERRORS**

### **Stand-Alone FORTRAN IV**

- T1 Construction
- T2 Usage
- T3 Data pool full
- T4 Illegal statement
- T5 Improper use of name
- T6 Improper statement number
- T7 Mode
- T8 Constant too large
- T9 Improper *DO* nesting

## MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual.

### OPTIONAL LISTINGS

Source and object records can be listed. Source records are listed as they are input. Object records are listed as they are created. Each object record consists of a varying number of two- and four-word data/instruction entries, one listing line per each entry. Two-word entries are of the form

```
abbc vvvvv
```

and four-word entries are of the form

```
abbc nnnnnn vvvvv
```

where

|        |                                           |
|--------|-------------------------------------------|
| a      | is the control code                       |
| bb     | is the subcode                            |
| c      | is the pointer number                     |
| nnnnnn | is a one- to six-letter subprogram name   |
| vvvvv  | is a six-digit octal value or instruction |

## MAPS

### Stand-Alone FORTRAN IV

#### RUNTIME

See figure 8-1.

#### PROGRAM

Upon processing the END statement, the compiler lists the program map. The first three lines of the map define the size of the program, data and common areas. They are of the form

```
a *SIZE mmmmmm
```

where

a is the area (0 = program, 1 = data)  
mmmmmm is the actual size

For programs with no termination errors, the following information is also listed.

**operation**

|    |       |              |                            |
|----|-------|--------------|----------------------------|
| a, | rrrrr | nnnnn        | Variable                   |
| a, | rrrrr | cccccc ccccc | Constant                   |
| S, | rrrrr | nnnnn        | External subprogram        |
| #, | rrrrr | ssss         | Statement number           |
| X, | rrrrr | ssss         | Undefined statement number |
| c, | rrrrr | nnnnn        | COMMON variable            |

where

|              |                                                                                                  |
|--------------|--------------------------------------------------------------------------------------------------|
| a            | is the area                                                                                      |
| rrrrr        | is the relative location of the item or the last reference to the subprogram or statement number |
| cccccc ccccc | is a two-word octal constant                                                                     |
| ssss         | is a statement number                                                                            |

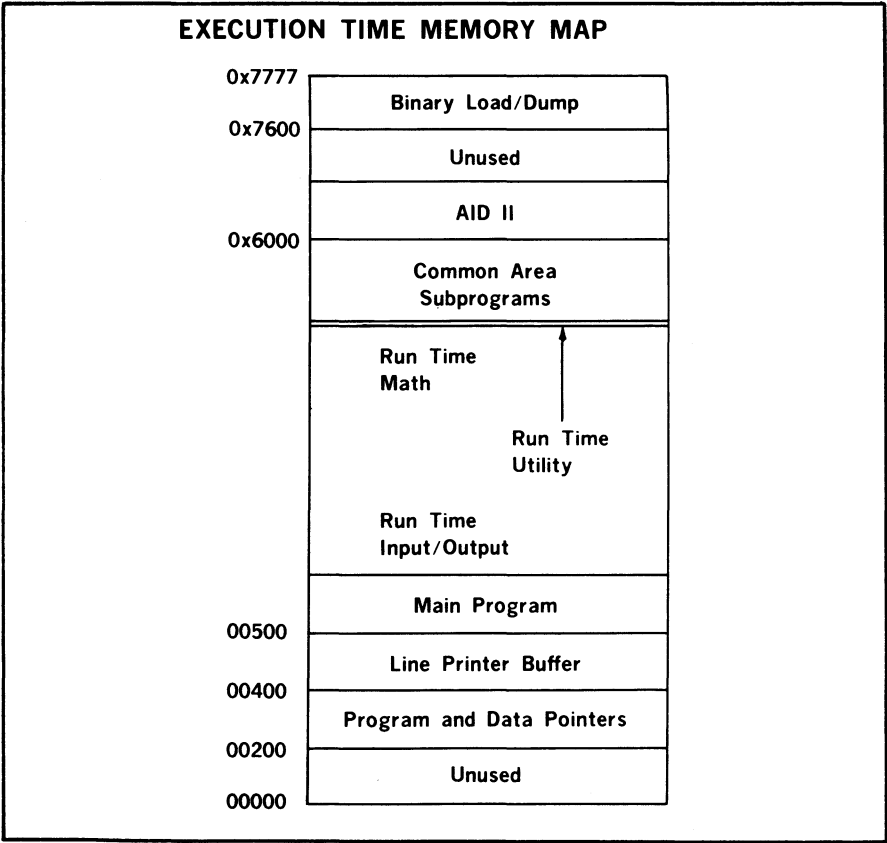


Figure 8-1. Run-Time Map for Stand-Alone FORTRAN IV

## MOS FORTRAN IV

### RUNTIME

See figures 2-1 and 2-2 in the Varian 620 Master Operating System Reference Manual.

### PROGRAM

In the following sample printout of an MOS FORTRAN IV program map, the leftmost six-digit column is the value of the relative location counter of the item to the right. The letter and name, value or variable identifies the item as a relocatable (R), fixed-, or floating-point value or variable, or a subroutine entry name, the name of an External (E) subprogram or the name of a region in COMMON (C). COMMON is the name for blank or unlabeled COMMON under MOS.

#### Sample Program Map

##### ENTRY/COMMON BLOCK NAMES

```
000043 R TEST1
000004 C COMMON
000004 C      XX
```

##### EXTERNAL NAMES

```
000020 E      $SI
```

##### SYMBOL TABLE

```
000023 R 000001
000027 R 000002
000000 C      A
000002 C      B
000000 C      C
000002 C      D
000025 R      I
000005 R      5
000031 R      J
000035 R      K
000033 R 000003
000012 R      10
000041 R      R
000037 R 040306 031463
000020 E      $ST
```

## I/O DEVICE SELECTION

### Stand-Alone FORTRAN IV

The following logical unit numbers are associated with the indicated devices at execution time.



## **operation**

| <b>Unit Number</b> | <b>Assignment</b>                    |
|--------------------|--------------------------------------|
| Logical unit 0     | Teletype keyboard and printer        |
| Logical unit 1     | Teletype paper tape reader and punch |
| Logical unit 2     | High-speed paper tape reader/punch   |
| Logical unit 3     | Card reader/punch                    |
| Logical unit 4     | Line printer                         |
| Logical unit 8     | Magnetic tape unit 0                 |
| Logical unit 9     | Magnetic tape unit 1                 |
| Logical unit 10    | Magnetic tape unit 2                 |
| Logical unit 11    | Magnetic tape unit 3                 |

## **MOS FORTRAN IV**

The logical unit names corresponding to the unit numbers, u, in READ, WRITE, and auxiliary I/O statements are:

| <b>Number</b> | <b>Name</b> | <b>Definition</b>    |
|---------------|-------------|----------------------|
| 1             | SF          | System file          |
| 2             | SI          | System input         |
| 3             | SO          | System output        |
| 4             | PI          | Processor input      |
| 5             | LO          | List output          |
| 6             | BI          | Binary input         |
| 7             | BO          | Binary output        |
| 8             | SS          | System scratch       |
| 9             | GO          | Load and go          |
| 10            | PO          | Processor output     |
| 11            | S1          | Scratch # 1          |
| 12            | S2          | Scratch # 2          |
| 13            | S3          | Scratch # 3          |
| 14            | S4          | Scratch # 4          |
| 15-255        | None        | User-definable files |

| Print<br>Character | Printer | Mag Tape<br>(7 trk. BCD) | ASCII | FORTRAN<br>(internal) | Hollerith<br>026 | 029      |
|--------------------|---------|--------------------------|-------|-----------------------|------------------|----------|
| (blank)            | 040     | 20                       | 240   | 00                    | no punch         | no punch |
| !                  | 041     | 52                       | 241   | 51                    | 11-2-8           | 11-2-8   |
| "                  | 042     | 35                       | 242   | 62                    | 0-5-8            | 7-8      |
| #                  | 043     | 37                       | 243   | 63                    | 0-7-8            | 3-8      |
| \$                 | 044     | 53                       | 244   | 60                    | 11-3-8           | 11-3-8   |
| %                  | 045     | 57                       | 245   | 64                    | 11-7-8           | 0-4-8    |
| &                  | 046     | 77                       | 246   | 65                    | 12-7-8           | 12       |
| '                  | 047     | 14                       | 247   | 66                    | 4-8              | 5-8      |
| (                  | 050     | 34                       | 250   | 52                    | 0-4-8            | 12-5-8   |
| )                  | 051     | 74                       | 251   | 53                    | 12-4-8           | 11-5-8   |
| *                  | 052     | 54                       | 252   | 47                    | 11-4-8           | 11-4-8   |
| +                  | 053     | 60                       | 253   | 45                    | 12               | 12-6-8   |
| ,                  | 054     | 33                       | 254   | 54                    | 0-3-8            | 0-3-8    |
| -                  | 055     | 40                       | 255   | 46                    | 11               | 11       |
| .                  | 056     | 73                       | 256   | 51                    | 12-3-8           | 12-3-8   |
| /                  | 057     | 21                       | 257   | 50                    | 0-1              | 0-1      |
| 0                  | 060     | 12                       | 260   | 01                    | 0                | 0        |
| 1                  | 061     | 01                       | 261   | 02                    | 1                | 1        |
| 2                  | 062     | 02                       | 262   | 03                    | 2                | 2        |
| 3                  | 063     | 03                       | 263   | 04                    | 3                | 3        |
| 4                  | 064     | 04                       | 264   | 05                    | 4                | 4        |
| 5                  | 065     | 05                       | 265   | 06                    | 5                | 5        |
| 6                  | 066     | 06                       | 266   | 07                    | 6                | 6        |
| 7                  | 067     | 07                       | 267   | 10                    | 7                | 7        |
| 8                  | 070     | 10                       | 270   | 11                    | 8                | 8        |
| 9                  | 071     | 11                       | 271   | 12                    | 9                | 9        |
| :                  | 072     | 15                       | 272   | 67                    | 5-8              | 2-8      |
| ;                  | 073     | 56                       | 273   | 70                    | 11-6-8           | 11-6-8   |
| <                  | 074     | 76                       | 274   | 76 <sup>2</sup>       | 12-6-8           | 12-4-8   |
| =                  | 075     | 13                       | 275   | 55                    | 3-8              | 6-8      |
| >                  | 076     | 16                       | 276   | 76 <sup>4</sup>       | 6-8              | 0-6-8    |
| ?                  | 077     | 72                       | 277   | 76 <sup>2</sup>       | 12-2-8           | 0-7-8    |
| @                  | 100     | 32                       | 300   | 77                    | 0-2-8            | 4-8      |
| A                  | 101     | 61                       | 301   | 13                    | 12-1             | 12-1     |
| B                  | 102     | 62                       | 302   | 14                    | 12-2             | 12-2     |
| C                  | 103     | 63                       | 303   | 15                    | 12-3             | 12-3     |
| D                  | 104     | 64                       | 304   | 16                    | 12-4             | 12-4     |
| E                  | 105     | 65                       | 305   | 17                    | 12-5             | 12-5     |
| F                  | 106     | 66                       | 306   | 20                    | 12-6             | 12-6     |
| G                  | 107     | 67                       | 307   | 21                    | 12-7             | 12-7     |
| H                  | 110     | 70                       | 310   | 22                    | 12-8             | 12-8     |

| Print<br>Character | Printer | Mag Tape<br>(7 trk. BCD) | ASCII | FORTTRAN<br>(internal) | Hollerith<br>026 | 029    |
|--------------------|---------|--------------------------|-------|------------------------|------------------|--------|
| I                  | 111     | 71                       | 311   | 23                     | 12-9             | 12-9   |
| J                  | 112     | 41                       | 312   | 24                     | 11-1             | 11-1   |
| K                  | 113     | 42                       | 313   | 25                     | 11-2             | 11-2   |
| L                  | 114     | 43                       | 314   | 26                     | 11-3             | 11-3   |
| M                  | 115     | 44                       | 315   | 27                     | 11-4             | 11-4   |
| N                  | 116     | 45                       | 316   | 30                     | 11-5             | 11-5   |
| O                  | 117     | 46                       | 317   | 31                     | 11-6             | 11-6   |
| P                  | 120     | 47                       | 320   | 32                     | 11-7             | 11-7   |
| Q                  | 121     | 50                       | 321   | 33                     | 11-8             | 11-8   |
| R                  | 122     | 51                       | 322   | 34                     | 11-9             | 11-9   |
| S                  | 123     | 22                       | 323   | 35                     | 0-2              | 0-2    |
| T                  | 124     | 23                       | 324   | 36                     | 0-3              | 0-3    |
| U                  | 125     | 24                       | 325   | 37                     | 0-4              | 0-4    |
| V                  | 126     | 25                       | 326   | 40                     | 0-5              | 0-5    |
| W                  | 127     | 26                       | 327   | 41                     | 0-6              | 0-6    |
| X                  | 130     | 27                       | 330   | 42                     | 0-7              | 0-7    |
| Y                  | 131     | 30                       | 331   | 43                     | 0-8              | 0-8    |
| Z                  | 132     | 31                       | 332   | 44                     | 0-9              | 0-9    |
| [                  | 133     | 75                       | 333   | 76 <sup>2</sup>        | 12-5-8           | 12-2-8 |
| \                  | 134     | 36                       | 334   | 76 <sup>2</sup>        | 0-6-8            | 11-7-8 |
| ]                  | 135     | 55                       | 335   | 76 <sup>2</sup>        | 11-5-8           | 0-2-8  |
| ↑                  | 136     | 17 <sup>1</sup>          | 336   | 76 <sup>2</sup>        | 7-8              | 12-7-8 |
| ←                  | 137     | 20                       | 337   | 76 <sup>3</sup>        | 2-8              | 0-5-8  |

1. End-of-file for magnetic tape (7 track).
2. Undefined character for Stand-alone FORTRAN IV compiler and runtime.
3. Teletype form control character for all FORTRAN IV compilers and runtimes (skip to column 1).
4. Teletype form control character for Stand-alone FORTRAN IV compiler and runtime (skip to column 7).

## 620 Standard Character Codes

# **BASIC Reference Manual**





# TABLE OF CONTENTS

## SECTION 1 A PRIMER IN BASIC

|                            |      |
|----------------------------|------|
| An Example .....           | 1-1  |
| Formulas .....             | 1-7  |
| Loops .....                | 1-11 |
| Arrays .....               | 1-13 |
| Errors and Debugging ..... | 1-16 |

## SECTION 2 ADVANCED BASIC

|                         |     |
|-------------------------|-----|
| Logical Operators ..... | 2-1 |
| Special Functions ..... | 2-2 |
| Matrices .....          | 2-4 |

## SECTION 3 STATEMENTS IN BASIC

|                                         |      |
|-----------------------------------------|------|
| READ and DATA Statements .....          | 3-2  |
| DIM (Dimension) Statement .....         | 3-3  |
| MAT (Matrix) Statement .....            | 3-3  |
| LET Statement .....                     | 3-3  |
| FOR and NEXT Statements .....           | 3-4  |
| IF/THEN Statement .....                 | 3-4  |
| GO TO Statements .....                  | 3-5  |
| GOSUB, RETURN, and SUB Statements ..... | 3-6  |
| PRINT Statements .....                  | 3-10 |
| INPUT Statement .....                   | 3-15 |
| RESTORE Statement .....                 | 3-16 |
| REM (Remark) Statement .....            | 3-17 |
| CALL Statement .....                    | 3-18 |
| WAIT Statement .....                    | 3-18 |
| STOP Statement .....                    | 3-18 |
| END Statement .....                     | 3-19 |

**SECTION 4**  
**USING THE BASIC SYSTEM**

|                                    |     |
|------------------------------------|-----|
| Operating Instructions .....       | 4-1 |
| Control Commands .....             | 4-3 |
| Program and Calculator Modes ..... | 4-4 |

**APPENDIX A**  
**ERROR MESSAGES**

**APPENDIX B**  
**CALL STATEMENT DESIGN CONSIDERATIONS**

**NOTE**

In the examples given in this manual, **boldface** type indicates obligatory items, and *italics* indicate optional items. Capital letters indicate precisely the letters used, and small letters indicate that other letters and or numbers are to be substituted.

## SECTION 1 — A PRIMER IN BASIC

A **program** is simply a set of directions that tells a computer how to solve a problem. The computer takes the raw-data input, manipulates it according to the directions in the program, and, if there are no errors in the data or program, gives the answer required.

For proper performance, any program must fulfill two requirements:

- a. *The program must be in a language that is understood by the computer.* just as problems presented to people must be in languages they understand.
- b. *The program must be complete and precise* because the computer, unlike human problem-solvers, cannot make inferences. The computer does what you order, not what you meant to order.

A program in English would pose insurmountable difficulties for the computer. English and other human languages are rich in ambiguities and redundancies, qualities that make poetry possible and computing impossible. Thus, you must present your program in a language that has many of the characteristics of ordinary mathematical notation: simple vocabulary and grammar, but with the ability to specify problem-solving steps completely and precisely. One such language is the **Beginner's All-purpose Symbolic Instruction Code (BASIC)**, originally developed at Dartmouth College.

In this manual, the rest of section 1 introduces you to the BASIC language and shows you how to write simple programs that can solve a wide variety of useful and interesting problems. Section 2 shows you how to apply the BASIC language to more advanced computer techniques. Section 3 shows you how to use and operate the BASIC system, and includes a variety of reference material.

### AN EXAMPLE

Let us begin by seeing how the BASIC language is applied to the solving of a system of two simultaneous linear equations in two variables, and then to the solving of two different systems, each differing from the first only in the constants c and f. Given:

$$ax + by = c$$

$$dx + ey = f$$

Then

$$x = (ce - bf)/(ae - bd)$$

$$y = (af - cd)/(ae - bd)$$



Note that, if  $ae - bd = 0$ , there is either no solution or an infinite number of solutions, but no solution that is unique. In any other case, there will be a unique solution.

Whether or not you understand the manual solution of such systems is not important. For now, study the following example and explanation to learn how a BASIC program for solving the problem is developed.

### EXAMPLE

```
10  READ A, B, D, E
15  LET G = A * E - B * D
20  IF G = 0 THEN 65
30  READ C, F
37  LET X = (C * E - B * F)/G
42  LET Y = (A * F - C * D)/G
55  PRINT X, Y
60  GO TO 30
65  PRINT "NO UNIQUE SOLUTION"
70  DATA 1, 2, 4
80  DATA 2, -7, 5
85  DATA 1, 3, 4, -7
90  END
```

Several things about the program can be noted:

- a. *It uses only capital letters* since the Teletype has only capital letters. A handwritten program separates confusing pairs of characters by adhering to the following conventions:

Numbers:    **O 1 2**

Letters:    **Ø I Z**

Since these characters are on different Teletype keys, there is no confusion during typing.

- b. *Each line of the program begins with a number* called a **line number** that identifies the line, called a **statement**, and specifies the order in which the statement is to be processed by the computer. The program can be written in any order since the computer will sort it and edit it as specified by the line numbers.
- c. *Each statement has a word following the line number.* This word specifies the **type of statement**.
- d. *Each statement is free form.* This is not obvious from the program printed above, but spaces have no effect on statements in BASIC. Thus, line 10 could have been typed as

```
10READA,B,D,E
```

and line 15 as

```
15LETG=A*B-D
```

The exception to this is the spacing in statements to be printed (e.g., line 65) since they print just as written, including spacing.

Now let us go through the example, statement by statement.

The first statement, line 10, is a READ statement. READ statements must be accompanied by one or more DATA statements, which do not, as the example shows, have to be adjacent to the READ statement in the program. Whenever the computer encounters a READ statement while executing a program, it assigns the next available values in the DATA statement(s) to the variables in the READ statement. Thus, the variable A in line 10 is assigned the value 1 from line 70, the first DATA statement. Similarly, B is assigned the value 2, D the value 4, and E the value next available in the DATA statements, i.e., the value 2 from line 80. The next READ statement will pick up DATA values from where this statement leaves off. This is further discussed below.

The second statement, line 15, is a LET statement. LET statements contain formulas to be evaluated using mathematical notation slightly modified to meet the requirements of the computer, e.g., use of the asterisk (\*), which cannot be omitted, to denote multiplication. In line 15, we order the computation of  $AE - BD$  and call the result G. In general, a LET statement directs the computer to set a single variable equal to a mathematical expression, where the variable is to the left of an equal sign and the mathematical expression to the right. *Note that in a LET statement the equal sign does not denote equality but replacement, and line 15 might best be read as "replace the quantity  $AE - BD$  with G"; thus, formulas such as  $X = X + 1$  are perfectly valid in LET statements.*

The next statement, line 20, is an IF/THEN statement. IF/THEN statements contain conditions that, if met, cause the execution of the program to go next to the statement designated after the THEN. In this example, we know that there can be no unique solution if  $G = 0$ , so we order the program to jump to line 65 if, and only if,  $G = 0$ .

The statement in line 65 is a PRINT statement. PRINT statements cause the output of the material between quotation marks and/or of computed values. Thus, if  $G = 0$  (line 20), the program prints the text

```
NO UNIQUE SOLUTION
```

and, since lines 70, 80, and 85 contain nonexecutable DATA statements, passes to line 90, an END statement. This terminates the execution of the program for the case where  $G = 0$ .

## a primer in basic

However, in our example,  $G \neq 0$ . Since the condition in line 20 is not met, the program continues to execute statements in the order of their appearance, rather than jumping to line 65. Thus, an IF/THEN statement tells the computer where to go for its next instruction when the IF condition is met, but, if the condition is not met, the computer passes to the next statement in sequence.

The next statement, line 30, is another READ statement that assigns the next two available DATA values, -7 and 5 (both from line 80), to the variables C and F, respectively. This supplies the rest of the required constants to the original pair of equations to yield the system

$$x + 2y = -7$$

$$4x + 2y = 5$$

The next statements, lines 37 and 42, are LET statements that direct the computer to find the values of X and Y according to the formulas provided. Note the use of parentheses to indicate that  $CE - BF$  is to be divided by G. Had the parentheses been omitted, the computer would have divided only BF by G, solving

$$x = ce - (bf/g)$$

rather than the required

$$x = (ce - bf)/g$$

that is equivalent to the original

$$x = (ce - bf)/(ae - bd)$$

The next statement, line 55, is a PRINT statement that causes the output of the computed values of X and Y. The values 4 and 5.5 are printed and the computer goes to the next statement.

The next statement, line 60, is a GO TO statement. A GO TO statement causes the execution of the program to go next to the statement designated. There is no condition. When the computer encounters a GO TO statement, it *always* goes to the statement designated.

The computer thus returns to line 30, which contains a READ statement. The next two available DATA values, 1 and 3 (from line 85), are assigned to the variables C and F, respectively. The old values of C and F are lost and replaced with the new values. This yields the system

$$x + 2y = 1$$

$$4x + 2y = 3$$

As before, the computer finds the values according to lines 37 and 42, prints the results as directed in line 55, and returns to line 30 as specified by line 60.

Here the next two values of C and F, 4 and -7, respectively (from line 85), are assigned and the computer solves the system

$$x + 2y = 4$$

$$4x + 2y = -7$$

It prints the solutions and returns to line 30. However, there are no more DATA values for assignment to C and F. The computer informs us that it is out of data by printing on the Teletype

### ERROR 56 IN LINE 30

(Error 56 does not indicate a *mistake* in the program, but merely conveys information from the computer concerning the lack of data. We will examine such **error messages** in detail later.)

This terminates the execution of the program. Note that it is not necessary to reach an END statement to terminate.

The program and the results are shown below just as they would appear on the Teletype. After typing in the program, type the word RUN and press the RETURN key. This directs the computer to execute the program.

#### Program

```

10  READ A, B, D, E
15  LET G = A*E - B*D
20  IF G = 0 THEN 65
30  READ C, F
37  LET X = (C*E - B*F)/G
42  LET Y = (A*F - C*D)/G
55  PRINT X, Y
60  GO TO 30
65  PRINT "NO UNIQUE SOLUTION"
70  DATA 1, 2, 4
80  DATA 2, -7, 5
85  DATA 1, 3, 4, -7
90  END

```

RUN

#### Result

```

4                -5.5
0.666666        0.166666
-3.66666        3.83333
ERROR 56 IN LINE 30

```

*Now that we have solved the problem, let us examine more closely certain characteristics of the program.*

For example, omission of line 20 would not have affected the solution just presented. However, in the case where  $ae - bd = 0$ , omission of line 20 would have required the computer to perform the impossible calculation of dividing by zero in lines 37 and 42. The computer would then merely print

**ERROR 69 IN LINE 37**

**ERROR 69 IN LINE 42**

where ERROR 69 indicates an attempt to divide by zero.

Omission of line 55 would have been catastrophic. The computer would have made the required calculations, but could not print them since it was not ordered to do so. The solutions would have remained the computer's secret. Furthermore, there would have been no error message to signal that something was amiss, since there was no error in the format of the program, the data, or the computations. The Teletype would have simply remained blank.

Omission of line 60 would not have affected the first set of solutions, but after the initial values of X and Y were printed, the computer would have gone to line 65, printed

**NO UNIQUE SOLUTION**

and stopped. At least the curious juxtaposition of a set of solutions followed by this message would have brought the error to our attention.

*The choice of individual line numbers is arbitrary, but the lines must be numbered in the order to be followed during the execution of the program* (disregarding jumps caused by GO TO and IF/THEN statements). We could have numbered the lines 1, 2, 3, ..., 13, although this is not recommended. Spacing between the numbers allows for insertions made necessary by omissions or by modifications. Thus, if we discover that we have left out two statements between lines 40 and 50, we can assign them numbers such as 44 and 46 so that the computer will place them in the proper order during the editing and sorting of the program.

*The division of data items among the DATA statements is arbitrary, but the items must appear in the order in which they are to be read during the execution of the program.* Thus, in our example, the first data item is assigned to the variable A, the second to B, the third to D, the fourth to E, the fifth to C, the sixth to F, the seventh to C (replacing the fifth), etc. Rather than the three lines 70, 80, and 85 given in the example, we might have written

**75 DATA 1, 2, 4, 2, -7, 5, 1, 3, 4, -7**

or perhaps more naturally

```

70  DATA 1, 2, 4, 2
75  DATA -7, 5
80  DATA 1, 3
85  DATA 4, -7

```

to indicate that the coefficients appear in the first DATA statement and the various pairs of values for C and F in the subsequent DATA statements.

## FORMULAS

The computer computes by evaluating the formulas in a program. These formulas are similar to those used in standard mathematical notation, with slight modifications required by the nature of the computer.

One general limitation in BASIC is that a formula must be written on a single line of the coding sheet or punched in a single card. However, long formulas can be broken down into two or more short formulas that meet this requirement. The limitation is thus one of writing and not one of computation.

## Operations

BASIC uses the following five *arithmetic operations* in formulas, each indicated by the corresponding symbol. Note that the asterisk (\*) used to indicate multiplication cannot be omitted, i.e., in ordinary mathematical notation  $AB$  is equivalent to  $A \times B$  or  $A \cdot B$ , but in BASIC this can be expressed only by  $A * B$ .

| Operation      | Symbol |         | Example                     |
|----------------|--------|---------|-----------------------------|
| Addition       | +      | $A + B$ | (Add B to A)                |
| Subtraction    | -      | $A - B$ | (Subtract B from A)         |
| Multiplication | *      | $A * B$ | (Multiply A by B)           |
| Division       | /      | $A / B$ | (Divide A by B)             |
| Exponentiation | ↑      | $A ↑ B$ | (Raise A to the power of B) |

BASIC also provides for evaluation of the following six *arithmetic relationships* in IF/THEN statements where such comparisons define the IF condition.

a primer in basic

| Relationship        | Symbol | Example |                                   |
|---------------------|--------|---------|-----------------------------------|
| Is equal to         | =      | A = B   | (A is equal to B)                 |
| Is not equal to     | #      | A # B   | (A is not equal to B)             |
| Is less than        | <      | A < B   | (A is less than B)                |
| Is greater than     | >      | A > B   | (A is greater than B)             |
| Is not less than    | > =    | A > = B | (A is equal to or greater than B) |
| Is not greater than | < =    | A < = B | (A is equal to or less than B)    |

The introductory example contained an instance of the use of an arithmetic relationship in the statement

```
20 IF G = 0 THEN 65
```

In addition to the arithmetic operations and relationships, the computer can evaluate several *mathematical functions* in BASIC:

| Function                                 |                                                                | Coding  |
|------------------------------------------|----------------------------------------------------------------|---------|
| Find the sine of x                       | } Where x is a number<br>or is an angle<br>measured in radians | SIN (X) |
| Find the cosine of x                     |                                                                | COS (X) |
| Find the tangent of x                    |                                                                | TAN (X) |
| Find the arctangent of x                 |                                                                | ATN (X) |
| Find e to the power of x                 |                                                                | EXP (X) |
| Find the natural logarithm of x (ln x)   |                                                                | LOG (X) |
| Find the absolute value of x (  x  )     |                                                                | ABS (X) |
| Find the square root of x ( $\sqrt{x}$ ) |                                                                | SQR (X) |
| Truncate x*                              |                                                                | INT (X) |

| Function               | Coding  |
|------------------------|---------|
| Randomize $x^*$        | RND (X) |
| Assign a sign to $x^*$ | SGN (X) |

\* These functions, which are not self-explanatory, are discussed in section 2.

In coding the above functions, we can substitute any number or mathematical expression for X. For example, to find the square root of  $(4 + x^3)$ , we would write

SQR (4+X^3)

## Parentheses

To ensure that items in formulas are properly grouped together for correct execution of a program, BASIC requires careful use of parentheses. Such use gives attention to the order in which the computer performs the calculations.

*An expression inside parentheses is computed before the quantity is used in further computation.* For example, in the expression  $A - (B + C)$ , the quantity  $B + C$  is first computed, and then the result subtracted from A. In the case of nested parentheses, the nests are computed from the inside out, e.g., in  $A - (B - (C + D))$ , the quantity  $C + D$  is computed first.

*In the absence of parentheses in an expression or part of an expression, the computer performs its calculations according to the following three levels of priority:*

- a. First, all exponentiation
- b. Next, multiplication and division
- c. Finally, addition and subtraction

*Within a level of priority, computations are performed from left to right.* For example, if we type  $A + B * C \uparrow D$ , the computer first raises C to the power D, multiplies the result by B, and then adds A to the product. If this is not the order intended, we specify the desired order by the use of parentheses: to raise the product of B and C to the power D before addition to A, we write  $A + (B * C) \uparrow D$ ; to multiply  $A + B$  by C and raise the product to the power D, we write  $((A + B) * C) \uparrow D$ .

**If there is any question in your mind about these priorities, add more parentheses to eliminate possible ambiguities.** Thus, the computer, faced with  $A - B - C$ , first subtracts B from A and then subtracts C from their difference. Faced with  $A / B / C$ , it first divides A by B, then divides that quotient by C. Given  $A \uparrow B \uparrow C$ , the computer raises A to the power B, then raises that result to the power C. Changing or clarifying this order of computation requires the use of parentheses.



Parentheses permit easy formulation of complicated mathematical expressions. Thus, to find the arctangent of the quantity

$$3x - 2e^n + 8$$

we write

```
ATN ( 3 * x - 2 * EXP ( N ) + 8 )
```

Or, to find the value of  $(5/8)^{1/7}$ , simply write the two-line BASIC program

```
10 PRINT ( 5 / 8 ) ^ 1 / 7
20 END
```

and the computer calculates the decimal form of the answer and prints it in less time than it took to type the program.

## Numbers

In BASIC, a **number** is a positive or negative decimal value of up to (approximately) seven significant digits. The following are thus valid numbers in BASIC: 2, -3.675, 1234567, -.7654321, and 483.4156.

The following are not valid numbers in BASIC: 14/3 and  $\sqrt{7}$ . These are expressions, not numbers, and must be converted by the computer to valid BASIC numbers before further manipulation or inclusion in a list of data. In the first case, the expression contains two numbers, 14 and 3, whose quotient in decimal form can be manipulated or included in a list of data. The second expression contains one number, 7, whose square root can be computed in BASIC by applying the appropriate mathematical function, SQR (7), to yield a valid decimal number that can be used in further computations or in a list of data.

Additional range and flexibility in the BASIC number system is attained by using the letter E (exponent), which is read "times ten to the power," e.g., 73E7 indicates 73 times 10 to the power 7. Thus, we can write 0.001234567 in several forms acceptable in BASIC, e.g., .1234567E - 2, 1234567E - 9, 1234.567E - 6. We can write ten million as 1E7 or 1E + 7, but we cannot write just E7 (this is a variable, section 1.2.4) since we must indicate 1 as the quantity that is to be multiplied by 10.

Numbers are processed in floating-point format; therefore, integer values can produce decimal approximations when results are output. For example, PRINT INT(8202) = 8202.1.

## Variables

In BASIC, a **variable** is denoted by a single letter or by a single letter followed by a single digit. The following are thus valid variables in BASIC: E7, A, X, I0, and Q2.

A variable in BASIC stands for a number, usually one whose value is not known when the

program is being written. Variables are assigned numerical values by LET, READ, and INPUT statements. Values thus assigned do not change until the next such statement containing a new value for that variable is encountered. Then, the new value replaces the former value and is used in subsequent manipulations.

*A numerical value must be assigned to a variable before it can be used in a computation* since all variables are undefined before each run. Failure to make such assignments results in the error message

**ERROR 50 IN LINE nn**

## LOOPS

Often programs contain portions to be performed repeatedly, sometimes with slight changes on each pass. Iterations and incrementations are examples of such cases. In BASIC, the **loop** simplifies the writing of such programs because the portion to be repeated is written only once.

For example, a BASIC program to print a table of the first 100 positive integers and their square roots would be 101 lines long without the use of loops:

```
10  PRINT 1, SQR (1)
20  PRINT 2, SQR (2)
30  PRINT 3, SQR (3)
.
.
.
990 PRINT 99, SQR (99)
1000 PRINT 100, SQR (100)
1010 END
```

However, the use of a loop reduces the 101-line program to five lines giving the same results:

```
10  LET X = 1
20  PRINT X, SQR (X)
30  LET X = X + 1
40  IF X <= 100 THEN 20
50  END
```

Line 10 assigns X the value of 1. Line 20 prints this value of X and its square root. Line 30 increases the value of X to 2 (the statement is read "let the new value of X be the old value of X plus one"). Line 40 asks if the new value of X is less than or equal to 100, and, since this is true, directs the computer back to line 20. Line 20 prints 2 and the decimal form of  $\sqrt{2}$ . Line 30 then increases the value of X to 3, line 40 returns the computer to line 20, etc., repeating the loop in this manner 100 times. On the 101st pass, however, the relationship in line 40 is false since X is now 101. Therefore, the computer does not return to line 20, but goes to line 50 and ends the program. This example shows the *four*

*characteristics of a loop:* initialization (line 10), body (line 20), modification (line 30), and exit test (line 40).

Because the necessity for the use of loops of the type just illustrated arises so often, BASIC provides FOR and NEXT statements to specify such a loop even more simply. Our sample program then reduces to

```
10  FOR X = 1 TO 100
20  PRINT X, SQR (X)
30  NEXT X
50  END
```

Line 10 specifies a range of values, and line 30 increments X and returns the computer to line 20. When the range specified in line 20 is exhausted, there is no "next X" and the computer goes to line 50 to terminate the program.

The above illustration shows the simplest form of the FOR statement where the variable is incremented by one on each pass. However, other increments can be specified by a STEP clause in the FOR statement. Thus, the above example can be modified to increment in steps of 5 by writing

```
10  FOR X = 1 TO 100 STEP 5
```

where the computer would assign 1 to X on the first pass, 6 on the second, 11 on the third, etc., up to 96. Since the next increment would be 101, which is out of the specified range, the program terminates after printing 96 and its square root.

STEP can be negative. The above example can be modified to print the table of square roots in reverse order by writing

```
10  FOR X = 100 TO 1 STEP -1
```

FOR statements can contain initial values, final values, and step sizes that are expressions of any required complexity. Thus, provided N and Z have been specified earlier in the program, we can write such statements as

```
99  FOR A = N + 7 * Z TO (Z - N)/3 STEP (N - 4 * Z)/10
```

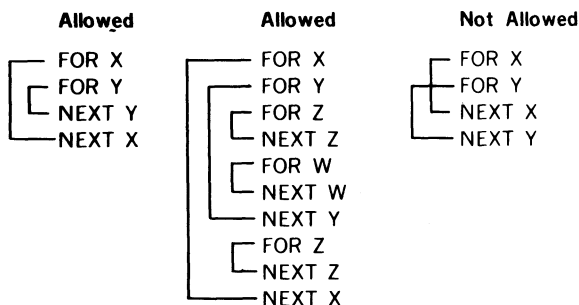
*The loop continues as long as the value of the control variable (i.e., the variable to the left of the equal sign) is within the specified range. This range can be specified by the conditions of an IF statement or by a FOR statement. If the initial value specified is already outside the range given in the IF statement, the body of the loop is not performed. The computer goes immediately to the line following the NEXT statement. Thus, in the following program for adding up the first n integers will give the correct result of zero when N is zero.*

```

10  READ N
20  LET S = 0
30  FOR K = 1 TO N
40  LET S = S + K
50  NEXT K
60  PRINT S
70  GO TO 10
90  DATA 3, 10, 0
99  END

```

Loops within loops are called **nested loops** and can be programmed with FOR and NEXT statements. However, nested loops must actually nest since loops that cross are invalid, as shown below.



## ARRAYS

BASIC can generate arrays by the use of **array variables** that consist of a single letter and are followed by subscripts in parentheses. The subscripts can be single, for example, to specify the coefficients of a polynomial ( $a_1, a_2, a_3, \dots$ ), or double, as in a two-dimensional matrix ( $b_{n,m}$ ). In BASIC, the first is written A(1), A(2), A(3), ... and the second B(1,1), B(1,2), ...

The letter used for an array variable can also be used as an ordinary BASIC variable (section 1.2.4) without confusion. However, within the same program, a letter cannot be used for both singly and doubly subscripted array variables.

The form of the subscript is flexible, and can be a number, variable, array variable, or mathematical expression. Thus, BASIC permits array elements such as B(1,K) and Q(A(3,7), B - C).

We can enter the one-dimensional array A(1), A(2), ..., A(10) into a program very simply:

```

10  FOR I = 1 TO 10
20  READ A(I)
30  NEXT I

```

```
40 DATA 2, 3, -5, 5, 2.2, 4, -9, 123, 4, -4
```

This simple form of array specification is all that is required for singly subscripted arrays in which no subscript is greater than ten. To specify larger arrays, use a DIM (dimension) statement of the form

```
DIM x(n)
```

where x is the array variable and n is the highest subscript in the array x. The DIM statement tells the computer to save sufficient space for the array. It is therefore advisable to allow for the maximum possible number of entries when the size of the array is not precisely known. For example, to enter a list of 15 numbers, we might write

```
10 DIM A(25)
20 READ N
30 FOR I = 1 TO N
40 READ A(I)
50 NEXT I
60 DATA 15
70 DATA 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
80 DATA 37, 41, 43, 47
```

Note that lines 20 and 60 could have been eliminated by writing FOR I = 1 TO 15, but the longer form given allows for lengthening the array up to 25 elements by changing only line 60. Of course, to extend the array beyond 25 elements requires, in addition, changing line 10.

Similarly, we can write the two-dimensional array B(1,1), B(1,2), ..., B(3,5) into a program as:

```
10 FOR I = 1 TO 3
20 FOR J = 1 TO 5
30 READ B(I,J)
40 NEXT J
50 NEXT I
60 DATA 2, 3, -5, -9, 2
70 DATA 4, -7, 3, 4, -2
80 DATA 3, -3, 5, 7, 8
```

No DIM statement is required as long as no subscript is greater than 10, i.e., the above entry could be expanded to include all entries up to B(10,10) without a DIM statement. An attempt to write an array with a subscript product greater than 100 without using a DIM statement yields the error message

```
ERROR 49 IN LINE xx
```

The error can be corrected by entering the missing DIM statement. To specify a 20-by-30

table in the above program, write

```
5      DIM B(20,30)
```

Since a DIM statement is merely a specification and is not executed during the program, it can be entered on any line before the END statement. However, it is convenient to place DIM statements near the beginning of the program.

The DIM statement is usually used to save more space than the ten subscripts automatically allowed by the computer, but in the special case of a long program containing many short arrays, DIM can be used to allot less space to arrays in order to leave more for the program.

Below is a program using both a singly and a doubly subscripted array. It computes the total sales of each of five salesmen, all of whom sell the same three products. Array P, specified in lines 10 through 30, gives the price-per-unit of each product. Array S, specified in lines 40 through 80, tells how many units of each product were sold by each salesman. The program reads the price data from line 900 into the elements of array P, and the sales data from lines 910 through 930 into the elements of array S. Thus, product 1 sells for \$1.25 per unit, product 2 for \$4.30, and product 3 for \$2.50; salesman 1 sold 40 units of the first product, 10 of the second, etc. To enter the sales for the next month and reuse the program for those data requires changing only lines 910 through 930. A price change requires a change to line 900.

### Program

```

      READY
10     FOR I = 1 TO 3
20     READ P(I)
30     NEXT I
40     FOR I = 1 TO 3
50     FOR J = 1 TO 5
60     READ S(I,J)
70     NEXT J
80     NEXT I
90     FOR J = 1 TO 5
100    LET S = 0
110    FOR I = 1 TO 3
120    LET S = S+P(I)*S(I,J)
130    NEXT I
140    PRINT "TOTAL SALES FOR SALESMAN" J; "$" S
150    NEXT J
900    DATA 1.25, 4.30, 2.5
910    DATA 40, 20, 37, 29, 42
920    DATA 10, 16, 3, 21, 8
930    DATA 35, 47, 29, 16, 33
999    END
RUN
```

### Result

```
TOTAL SALES FOR SALESMAN 1 $180.5
TOTAL SALES FOR SALESMAN 2 $211.3
TOTAL SALES FOR SALESMAN 3 $131.65
TOTAL SALES FOR SALESMAN 4 $166.55
TOTAL SALES FOR SALESMAN 5 $169.4
```

## ERRORS AND DEBUGGING

More often than not, the first running of a program will reveal **errors**. Program errors are of two types:

- a. A **format (grammatical) error** produces an error message (appendix A) that informs you of the type and location of the mistake so that you can correct it easily.
- b. A **logical error** does not produce an error message since it does not have any characteristics that appear to the computer as mistakes. A logical error can, however, result in incorrect results, or no results at all. The logical error is thus more difficult to detect and correct than the format error, particularly when the results seem somewhat reasonable.

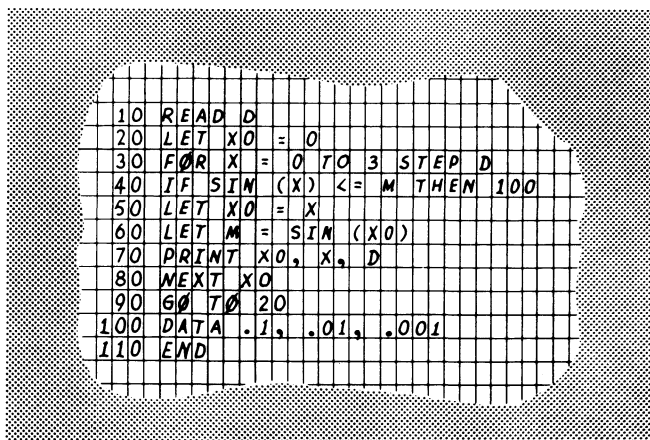
In either case, the error is first isolated and then it is corrected by inserting, deleting, or changing lines in the program.

- a. *To insert a line in a program*, type the new line using a line number between the line numbers of the lines that are to precede and follow the new line; e.g., to insert a line between lines 70 and 80, type the new line with a line number of 75.
- b. *To delete a line from a program*, type only the line number and then press RETURN.
- c. *To change a line*, retype the line correctly using the same line number.

Such corrections can be made at any time because the computer sorts the line numbers automatically.

The rest of this section is an illustration of the process of **debugging** (correcting) a program based on the problem of finding that value of  $x$  between 0 and 3 for which the sine of  $x$  is a maximum, and ask the machine to print out this value of  $x$  and the value of its sine. If you have studied trigonometry, you know that  $\pi/2$  is the correct value, but we shall use the computer to test successive values of  $x$  from 0 to 3, first using intervals of 0.1, then of 0.01, and finally of 0.001. Thus, we ask the computer to find the sine of 0, of 0.1, of 0.2, of 0.3, ..., of 2.8, of 2.9, and of 3, and to determine which of these 31 values is the largest. The computer does this by testing SIN(0) and SIN(0.1) to determine which is

larger, and calls the larger of these two numbers M. It then picks the larger of M and SIN(0.2) and this value is called M. This number is checked against SIN(0.3), etc. Each time a larger value of M is found, the value of x is remembered in X0. When the computer completes this process, M will have been assigned to the largest value. The search is repeated; this time the computer checks the 31 numbers (0, 0.01, 0.02, 0.03, ..., 2.98, 2.99, and 3), finding the sine of each and determining which is the largest. After this, the computer makes a third run using increments of 0.001. At the end of these three runs, the computer is to print three values: (1) the value of X0 that has the largest sine, (2) the sine of that value, and (3) the interval used on the run that found the value. To solve this problem, we begin by writing the program:



Now we are ready to enter the program on the Teletype. The sequence shown below is the entire sequence that appears on the Teletype, followed by explanatory comments.

```

READY
10 READ D
20 LWR X0 = 0
ERROR 11 IN LINE 20

20 LET X0 = 0
30 FOR X = 0 TO 3 STEP D
40 IF SINE - (X) <= M THEN 100
50 LET X0 = X
60 LET M = SIN (X)
70 PRINT X0, X, D
ERROR 36 IN LINE 70

70 PRINT X0, X, D
80 NEXT Z-X0
90 GO TO 20

```



```
100 DATA .1, .01, .001
110 END
RUN
ERROR 40 IN LINE 30
```

**Comments:** *Line 20:* The error message indicates that LET is mistyped. The correction is made by retyping the line correctly.

*Line 40:* The letter E is incorrectly entered after SIN. Typing a backarrow deletes the character immediately preceding. When an error is caught early enough to use this simple corrective technique, it is employed. The incorrect character is then replaced by the correct one (in this case, a blank) by typing it after the backarrow.

*Line 70:* The second error message indicates that XO is used for a variable rather than the correct form X0. The correction is made by retyping the line correctly.

*Line 80:* Another use of the backarrow replaces the incorrect character Z with the correct character X.

*Last line:* After typing the END statement, we try to run the program by typing RUN. However, this causes another error message, this time to advise us that the program contains a FOR statement without a corresponding NEXT. Upon checking, we see that this is caused by having the variables in lines 40 and 80 different. This is corrected by retyping line 80 using the same variable as that found in line 40. Furthermore, the IF/THEN statement in line 40 directs the computer to a DATA statement instead of to line 80. Line 40 is thus also retyped and another attempt made to run the program.

```
80 NEXT X
40 IF SIN (X) <= M THEN 80
RUN
ERROR 50 IN LINE 40
```

The new error message indicates that M has never been assigned an initial value. (This error came to the attention of the computer only after line 40 had been corrected.) We assign M an initial value of -1 and make another attempt to run the program.

```
20 LET M = -1
RUN
```

#### Result

```
0      0      .1
.1     .1     .1
.2     .2     .1
.3     .3     .1
.4     .4     .1
READY
```

We are now getting results, but they are incorrect. We are receiving every value of X, X0, and interval size. Therefore, the printout is stopped by typing any character on the Teletype while it is running. This error is corrected by typing:

```
70
85 PRINT X0, M, D
RUN
```

to move the PRINT statement outside the loop. Typing line number 70 followed by a carriage return deletes that line. It is retyped using line number 85 and with the incorrect variable X replaced by the correct variable M.

This attempt to run the program yields the following results:

```
1.59999    .999573    .1
1.59999    .999573    .1
1.59999    .999573    .1
1.59999    .999573    .1
READY
```

because line 90 returns the computer to line 20, merely repeating the operation using the same values, rather than to line 10 to pick up a new value for D. At the same time that we make this correction, we decide to add headings for the columns of figures of the results. Thus, we type:

```
90 GO TO 10
5 PRINT "X VALUE", "SIN", RESOLUTION"
ERROR 21 IN LINE 5
```

The new error message indicates the format error in line 5, in which there is no left quotation mark for the third item. We retype line 5 correctly and run the program.

```
5 PRINT "X VALUE", "SIN", "RESOLUTION"
RUN
```

#### Result

```
X VALUE    SIN        RESOLUTION
1.59999    .999573    .1
1.56998    .999999    1.00000E-02
1.5709     .999999    1.00000E-03
ERROR 56 IN LINE 10
```

Thus, we now obtain the correct results as specified in the original problem. (Remember that ERROR 56 does not indicate a mistake, but merely that there are no more data.) Having changed so many parts of the program, we request a list of the corrected program. Periodic listing of the present state of the program is an important debugging aid. The listing is requested merely by typing LIST. Typing PLIST at the end of the listing causes

the program to be punched on paper tape for later use. The correct listing appears on the Teletype as:

```
LIST
5    PRINT "X VALUE", "SIN", "RESOLUTION"
10   READ D
20   LET M = -1
30   FOR X = 0 TO 3 STEP D
40   IF SIN (X) <= M THEN 80
50   LET X0 = X
60   LET M = SIN (X)
80   NEXT X
85   PRINT X0, M, D
90   GO TO 10
100  DATA .1, 1.00000E-02, 1.00000E-03
110  END
PLIST
```

One common debugging aid that we have not used is the insertion of a PRINT statement to check that the computer is doing what we think we asked it to do. For example, if we wondered about the computation of M in the above example, we could have inserted 65 PRINT M to have the values of M printed.

## SECTION 2 – ADVANCED BASIC

This section explains the following specialized aspects of the BASIC language:

- Logical Operators
- Special Functions
- Matrices

### LOGICAL OPERATORS

In addition to the arithmetic operations and relationships, and mathematical functions already discussed (section 1.2.1), BASIC uses the **Boolean logical operators** AND, OR, and NOT. With the exception of NOT, which takes the following operand as its single argument, all of these are binary operators. In a formula, the binary operators have a lower priority than any of the arithmetic operators (i.e., \*, /, +, and -). Among the binary operators, the priority in ascending order is OR, AND, and the relational operators (all of equal priority). NOT has the same priority as unary + and unary - . Thus,

$$\text{NOT } A + B = C \text{ OR SGN}(K) \text{ AND SGN}(J)$$

is equivalent to

$$(((\text{NOT } A) + B) = C) \text{ OR } (\text{SGN}(K) \text{ AND } \text{SGN}(J))$$

The relational operators take algebraic numbers as arguments and return 0 (false) or 1 (true) according to the relationship existing between their arguments. Thus,  $3 < 2$  evaluates to 0 and  $A \neq 0$  evaluates to 1 if A has a nonzero value. The Boolean operators consider their arguments as zero (false) or nonzero (true) and return 0 and 1 as follows:

#### AND

| Argument 1 | Argument 2 | Result |
|------------|------------|--------|
| nonzero    | nonzero    | 1      |
| nonzero    | zero       | 0      |
| zero       | nonzero    | 0      |
| zero       | zero       | 0      |

**OR**

| Argument 1 | Argument 2 | Result |
|------------|------------|--------|
| nonzero    | nonzero    | 1      |
| nonzero    | zero       | 1      |
| zero       | nonzero    | 1      |
| zero       | zero       | 0      |

**NOT**

| Argument | Result |
|----------|--------|
| nonzero  | 0      |
| zero     | 1      |

Thus, 3 AND 1 evaluates to 1 while NOT -3 AND 0 evaluates to 0. It is important to realize the  $A < B < C$  is not equivalent to  $A < B$  AND  $B < C$ . If  $A = -3$ ,  $B = -2$ , and  $C = 1/2$ , the former evaluates as  $(-3 < -2) < 1/2$  or 0 (false), whereas the latter evaluates as  $(-3 < -2)$  AND  $(-2 < 1/2)$  or 1 (true).

**SPECIAL FUNCTIONS**

This section explains the special functions INT, RND, and SGN (listed in section 1.2.1), and the use of DEF to define other functions.

**INT (Integer) Function**

The INT function computes the value of  $x$  expressed by the algebraic notation as  $[x]$ . It gives the greatest integer not greater than  $x$  for  $-32768 \leq x < 32768$ . Thus,  $\text{INT}(2.35) = 2$ ,  $\text{INT}(-2.35) = -3$ , and  $\text{INT}(12) = 12$ . Note that  $\text{INT}(X) = 32768$  for  $x \geq 32768$  and  $\text{INT}(X) = -32768$  for  $x \leq -32768$ .

One use of the INT function is to truncate numbers. Use it to truncate to the nearest integer by writing  $\text{INT}(X + .5)$ . This will truncate 2.9, for example, to 3 by finding  $\text{INT}(2.9 + .5) = \text{INT}(3.4) = 3$ . Thus, this function will truncate a number midway between two integers, up to the larger of the integers.

It can also be used to round to any specific number of decimal places. For example,  $\text{INT}(10 * X + .5)/10$  rounds to one decimal place, and  $\text{INT}(10 * D \uparrow X + .5)/10 \uparrow D$  rounds to  $d$  decimal places.

**RND (Randomize) Function**

The RND function produces a normal distribution of random numbers between 0 and 1. The form of RND,  $\text{RND}(X)$  or  $\text{RND}(0)$ , requires an argument, although the argument has no significance. The argument can be a constant or a previously defined variable. The example below produces 20 random six-digit decimals.

**Program**

```

READY
10  FOR L = 1 TO 20
20  PRINT RND(0);
30  NEXT L
40  END
    RUN

```

**Result**

```

RUN
6.80170E-02 .240643 .417191 .192204 .701485 .705105
1.80673E-02 .460499 .87426 .34657 .812546 .146031
.723061 .473629 .249848 .182393 .697106 1.98793E-02
.122941 .221928  READY

```

```

RUN
6.80170E-02 .240643 .417191 .192204 .701485 .705105
1.80673E-02 .460499 .87426 .34657 .812546 .146031
.723061 .473629 .249848 .182393 .697106 1.98793E-02
.122941 .221928  READY

```

Note that the second RUN gives exactly the same random numbers as the first. This greatly facilitates the debugging of programs that use the random-number generator.

To produce 20 random one-digit integers, change line 20 to

```

20  PRINT INT(10*RND(0)),

```

**Result**

```

0   2   4   1   7   7   0   4   8   3   8   1
7   4   2   1   6   0   1   2  READY

```

To vary the type of random numbers (for example, to obtain 20 random numbers ranging from 1 to 9 inclusive), change line 20 to

```

20  PRINT INT(9*RND(0) + 1);

```

**Result**

```

1   3   4   2   7   7   1   4   8   4   8   2
7   5   3   2   7   1   2   2  READY

```

To obtain random numbers that are integers between 5 and 24 inclusive, change line 20 to

```
20 PRINT INT(20*RND(0) + 5);
```

### Result

```
6      9      13      8      19      19      5      14      22      11      21      7
19      14      9      8      18      5      7      9      READY
```

## SGN (Sign) Function

The SGN function assigns the value 1 to any positive number, 0 to zero, and -1 to any negative number. Thus,  $\text{SGN}(7.23) = 1$ ,  $\text{SGN}(0) = 0$ , and  $\text{SGN}(-.2337) = -1$ .

## DEF (Define) Function

In addition to standard functions, any other function can be defined with DEF. The name of the defined function comprises three letters, the first two of which are FN. A total of 26 functions can be defined, e.g., FNA, FNB, etc.

For example, DEF can be used in a program where the function  $\exp(-x^2 + 5)$  is needed frequently:

```
30 DEF FNE(X) = EXP(-X^2+5)
```

Various values of the function can be called by writing FNE(.1), FNE(3.45), FNE(A+2), etc. Such a definition can be a great time-saver to produce values of some function for a number of different values of the variable.

DEF can occur anywhere in the program, and the expression to the right of the equal sign can be any legal expression. It can contain a combination of other functions, including those defined by other DEF statements. It can involve variables other than the one denoting the argument of the function.

For example, assume FNR is defined by

```
70 DEF FNR(X) = SQR(2+LOG(X))-EXP(Y*Z)*(X+SIN(2*Z))
```

If values have been previously assigned to Y and Z, FNR(2.7) can be requested. New values can be assigned to Y and Z before the next use of FN.

The use of DEF is generally limited to those functions whose values can be computed within a single BASIC statement. More complicated functions or parts of a program are coded as subroutines accessible to GOSUB statements (section 3.8).

## MATRICES

It is often convenient to interpret doubly subscripted arrays as **matrices**. Although matrix computations can be worked out using conventional BASIC statements, the language provides the following 12 matrix (MAT) statements to simplify program writing and

increase the power of the language.

|                          |                                                                                                                 |
|--------------------------|-----------------------------------------------------------------------------------------------------------------|
| <b>MAT c = ZER</b>       | Fill matrix c with zeros                                                                                        |
| <b>MAT c = CON</b>       | Fill matrix c with ones                                                                                         |
| <b>MAT c = IDN</b>       | Define c as an identity matrix                                                                                  |
| <b>MAT PRINT a, b; c</b> | Print three matrices, in this example with a and c in the regular format and b closely packed (section 3.9.2)   |
| <b>MAT b = a</b>         | Set matrix b equal to matrix a                                                                                  |
| <b>MAT c = a + b</b>     | Add the two matrices a and b and place the result in matrix c                                                   |
| <b>MAT c = a - b</b>     | Subtract matrix b from matrix a and place the result in matrix c                                                |
| <b>MAT c = a * b</b>     | Multiply matrix a by matrix b and place the result in matrix c                                                  |
| <b>MAT c = TRN(a)</b>    | Transpose matrix a and place the result in matrix c                                                             |
| <b>MAT c = (k) * a</b>   | Multiply matrix a by the number or expression k (which must be in parentheses) and place the result in matrix c |
| <b>MAT c = INV(a)</b>    | Invert matrix a and place the result in matrix c                                                                |

*BASIC matrices adhere to the following convention: if a MAT statement specifies a matrix having the dimensions m-by-n, the rows are numbered 1, 2, ..., M and the columns 1, 2, ..., N.*

MAT statements are used in conjunction with dimension (DIM) statements that indicate the *maximum dimensions* of the matrices and cause the computer to save sufficient space. (The assumed dimensions of 10 rows by 10 columns for matrices without DIM statements do not apply for matrices involved in matrix computations. These must always be described by a DIM statement.) For example, to save space for any matrix up to and including 20 rows and 35 columns, we write

**DIM M(20,35)**



| Element Position | Memory Location | Element |
|------------------|-----------------|---------|
| 4                | m + 6           | A(1,2)  |
| 5                | m + 8           | A(2,2)  |
| 6                | m + 10          | A(3,2)  |
| 7                | m + 12          | A(1,3)  |
| 8                | m + 14          | A(2,3)  |
| 9                | m + 16          | A(3,3)  |

Given the statement DIM A(M,N), the location m of any element A(i,j) of the matrix, with respect to the first element A(1,1), is given by:

$$m = [\text{location of } A(1,1)] + 2[M(i - 1) + (j - 1)]$$

The three statements:

```
MAT M = ZER
MAT M = CON
MAT M = IDN
```

set up the matrix M filled with zeros, filled with ones, or as an identity matrix, respectively. Each acts as MAT READ as far as the dimensioning of the matrix is concerned. For example,

```
MAT M = CON( 7 , 3 )
```

sets up a 7-by-3 matrix full of ones, but

```
MAT M = CON
```

sets up a matrix, also full of ones, according to dimensions previously specified by a DIM statement. Thus,

The *actual dimensions* of a matrix can be defined either when first established (by using a DIM statement), or by one of the four matrix statements MAT READ, MAT ZER, MAT CON, or MAT IDN. Thus, to read a 20-by-7 matrix for x, write

```
10  DIM X( 20 , 7 )
    .
    .
    .
50  MAT READ X
```

To read a 17-by-30 matrix for y within maximum dimensions of 20-by-35, write

```

10  DIM Y(20,35)
    .
    .
    .
50  MAT READ Y(17,30)

```

The elements of a matrix are stored by column in ascending locations in memory, using two computer words for each element. Thus, the matrix dimensioned as DIM A(3,3) is structured and stored as follows:

|      |  | Columns |        |        |
|------|--|---------|--------|--------|
| Rows |  | A(1,1)  | A(1,2) | A(1,3) |
|      |  | A(2,1)  | A(2,2) | A(2,3) |
|      |  | A(3,1)  | A(3,2) | A(3,3) |

The elements would be stored in the following order:

| Element Position | Memory Location | Element |
|------------------|-----------------|---------|
| 1                | m               | A(1,1)  |
| 2                | m + 2           | A(2,1)  |
| 3                | m + 4           | A(3,1)  |

```

10  DIM M(20,7)
20  MAT READ M(7,3)
    .
    .
    .
35  MAT M = CON
    .
    .
    .
70  MAT M = ZER(15,7)

```

will first read in a 7-by-3 matrix for M and then set up a 7-by-3 matrix of ones for M as specified in line 20. This results in an error message because line 70 calls for 105 components in a matrix limited to 21 components by line 20. The original dimensions can be exceeded, however, provided the total number of components is within the set limit, e.g.,

```

90  MAT M = ZER(25,5)

```

The MAT PRINT statement (see also section 3.9) prints matrix components row by row across the page. Spacing between elements is controlled by typing commas or semicolons after each component, where commas space the printing and semicolons close-pack it. Each row starts on a new line. Rows containing more components than can be printed on one line are continued on the next line. Thus, the statement

```
MAT PRINT A, B; C
```

prints the matrices A and C in the normal format of five components per line, and matrix B closely packed with up to 12 components per line.

## Vectors

A singly subscripted array can be interpreted as a column vector. Vectors can be used in place of matrices as long as the above rules are followed. Since a vector like V(J) is treated as a column vector by BASIC, a row vector must be specified as a matrix with one row, e.g.,

```
DIM X(7), Y(1,5)
```

introduces a seven-component column vector and a five-component row vector. A column vector is printed one element per line with double spacing between lines. A row vector is printed as specified by the statement, e.g., where V is a row vector,

```
MAT PRINT V,
```

prints V as a row vector, five components to the line, while

```
MAT PRINT V;
```

prints V as a row vector, twelve components to the line.

## Manipulating Matrices

To set up a matrix B identical to the matrix A, provided that the dimensions previously assigned to B are the same as those of A, write

```
MAT B = A
```

If matrices A, B, and C have the same dimensions, operations such as

```
MAT C = A + B  
MAT C = A - B
```

are legal. The indicated operation is performed and stored in C. Only one operation per statement is allowed, e.g., to perform  $\text{MAT D} = \text{A} + \text{B} - \text{C}$  two statements are required.

In the multiplication operation

```
MAT C = A * B
```

the number of columns in A is equal to the number of rows in B, the number of rows in C is equal to the number of rows in A, and the number of columns in C is equal to the number of columns in B. For example, if A is l-by-m, and B is m-by-n, then C is l-by-n.

(Note that even when  $\text{MAT } A = A + B$  is legal,  $\text{MAT } A = A * B$  results in nonsense because, in multiplying matrices, components required to complete the computation are destroyed before they can be used, whereas, in addition, the results are stored immediately. However,  $\text{MAT } B = A * A$  is legal provided  $A$  is a square matrix.)

Matrices can also be multiplied by constants. In the operation

```
MAT C = (k) * A
```

each component of the matrix  $A$  is multiplied by  $k$  to form the components of the matrix  $C$ . The constant  $k$ , which is enclosed in parentheses, can be a number or an expression. The statement  $\text{MAT } A = (k) * A$  is legal.

To transpose the matrix  $A$ , write

```
MAT C = TRN(A)
```

where matrix  $C$  is matrix  $A$  transposed. Thus, if  $A$  is  $m$ -by- $n$ ,  $C$  is  $n$ -by- $m$ . Since a matrix is destroyed by transposition, it cannot be transposed into itself, i.e., dimensions cannot be reversed by writing  $\text{MAT } A = \text{TRN}(A)$ .

To invert the square matrix  $A$ , write

```
MAT C = INV(A)
```

where matrix  $C$  is the inversion of the square matrix  $A$ . Like transposition, a matrix cannot be inverted into itself.

## Sample Matrix Programs

**EXAMPLE 1:** This program reads in  $A$  and  $B$  in line 30 and in so doing sets up the correct dimensions. Dimensions for  $C$  are set up in line 35. Then, in line 40,  $A + A$  is computed and the answer is called  $C$ . Note that the data in line 90 result in  $A$  being 2-by-3 and  $B$  being 3-by-3. Both **MAT PRINT** formats are illustrated and one method of labeling a matrix print is shown.

### Program

```
READY
10  DIM A(15,15),B(15,15),C(15,15)
20  READ M, N
30  MAT READ A(M,N),B(N,N)
35  MAT C = ZER(M,N)
40  MAT C = A + A
50  MAT PRINT C;
60  MAT C = A*B
70  PRINT
```

```

75  PRINT "A*B="
80  MAT PRINT C,
90  DATA 2, 3
91  DATA 1, 2, 3
92  DATA 4, 5, 6
93  DATA 1, 0, -1
94  DATA 0, -1, -1
95  DATA -1, 0, 0
99  END

```

**Result**

```

      RUN
      2      4      6
      8      10     12
A*B =
     -2     -2     -3
     -2     -5     -9

```

**EXAMPLE 2:** This program inverts an n-by-n Hilbert matrix:

|     |       |       |     |       |
|-----|-------|-------|-----|-------|
| 1   | 1/2   | 1/3   | ... | 1/n   |
| 1/2 | 1/3   | 1/4   | ... | 1/n+1 |
| 1/3 | 1/4   | 1/5   | ... | 1/n+2 |
| ⋮   | ⋮     | ⋮     | ⋮   |       |
| ⋮   | ⋮     | ⋮     | ⋮   |       |
| ⋮   | ⋮     | ⋮     | ⋮   |       |
| 1/n | 1/n+1 | 1/n+2 |     | 1/n-1 |

Ordinary BASIC instructions are used to set up the matrix in lines 50 to 90. Note that this occurs after correct dimensions have been declared. Then a single instruction results in the computation of the inverse matrix, and one more instruction prints it. In this example, we have supplied 4 for n in the DATA statement and have made a run for this case.

**Program**

```

READY
5    REM THIS PROGRAM INVERTS AN N-BY-
      N HILBERT MATRIX
10   DIM A(20,20), B(20,20)
20   READ N
30   MAT A = CON(N,N)
40   MAT B = CON(N,N)
50   FOR I = 1 TO N
60   FOR J = 1 TO N
70   LET A(I,J) = 1/(I+J-1)

```

```

80  NEXT J
90  NEXT I
100 MAT B = INV(A)
110 PRINT
115 PRINT "INV(A) ="
120 PRINT
125 MAT PRINT B;
190 DATA 4
199 END
RUN

```

### Result

```

INV(A) =
15.9885   -119.877   239.713   -139.816
-119.877   1198.69   -1696.94   1678.04
239.713   -2696.94   6472.84   -4195.42
-139.816   1678.04   -4195.42   2797.07

```

**Note:** Because of severe rounding errors, Hilbert matrices are not inverted beyond  $n = 7$ .



## SECTION 3 – STATEMENTS IN BASIC

This section explains each type of BASIC statement and illustrates its use:

- READ statement
- DATA statement
- DIM (dimension) statement
- MAT (matrix) statements
- LET statement
- FOR statement
- NEXT statement
- IF/THEN statement (conditional GO TO statement)
- GO TO statements
  - Unconditional GO TO statement
  - Computed GO TO statement
- GOSUB (go to subroutine) statements
  - Unconditional GOSUB statement
  - Computed GOSUB statement
  - GOSUB statement with parameters
- RETURN statement
- SUB (subroutine) statement
- PRINT statement
- INPUT statement
- RESTORE statement
- REM (remark) statement



## statements in basic

- CALL statement
- WAIT statement
- STOP statement
- END statement

## READ and DATA Statements

The READ statement has the format

**number        READ var,var,...**

and the DATA statement has the format

**number        DATA num,num,...**

where

|            |               |
|------------|---------------|
| <b>var</b> | is a variable |
| <b>num</b> | is a number   |

The sequences of variables and numbers in these statements can contain any number of items as long as the statement itself does not exceed 72 characters.

A READ statement assigns values obtained from a DATA statement to the listed variables. Neither statement is used without one of the other type. A READ statement causes the variables listed in it to be given, in order, the next available numbers in the collection of DATA statements. Before the program is run, the computer takes all of the DATA statements in the order in which they are numbered and creates a large data block. Each time a READ statement is encountered anywhere in the program, the data block supplies the next available number or numbers. If the data block runs out of data with a READ statement still asking for more, the program is assumed to be done and we get an OUT OF DATA error message:

**ERROR 56 IN LINE nn**

Since we have to read in data before working with it, READ statements normally occur near the beginning of a program. The location of DATA statements is arbitrary, as long as they are numbered in the correct order. A common practice is to collect all DATA statements and place them just before the END statement.

### Examples:

```
150  READ X, Y, Z, X1, Y2, Q9
330  DATA 4, 2, 1.7
340  DATA 6.734E-3, -174.321, 3.14159265
```

```

234  READ B(K)
263  DATA 2, 3, 5, 7, 9, 11, 10, 8, 6, 4
10   READ R(I,J)
440  DATA -3, 5, -9, 2.37, 2.9876, -437.234E-5
450  DATA 2.765, 5.5576, 2.3789E2

```

Remember that only numbers are put in a DATA statement, and that  $15/7$  and  $\sqrt{3}$  are expressions, not numbers.

## DIM (Dimension) Statement

The DIM statement has the format

**number          DIM array**

where

**number**          is the line number of the statement  
**array**            is the name of the array being dimensioned  
                      followed by its subscript(s) in parentheses

The DIM statement is required for defining an array having any subscript greater than 10. The maximum subscript allowed is 255.

**Examples:**

```

20   DIM H (35)
35   DIM Q (5,25)

```

## MAT (Matrix) Statement

The MAT statement is explained in the section on matrices (section 2.3).

## LET Statement

This statement has the format

**number          LET var = exp**

or the format

**number          LET var = voa = ... = exp**

where

**number**          is the line number of the statement  
**var**                is a variable  
**exp**                is a number or an expression  
**voa**                is a variable or an array

This statement assigns the value of the number or expression to one or more variables or arrays.

## statements in basic

### Examples:

```
100 LET X = X + 1
200 LET W7 = (W-X4+3)*(Z-A/(A-B))-17
333 LET X = Y3 = A(3,1) = 1
900 LET W = Z = 3*X-4*X+2
```

## FOR and NEXT Statements

The FOR statement has the format

**number**      **FOR** **var** = **expi** **TO** **expf** **STEP** **exps**

and the NEXT statement has the format

**number**      **NEXT** **var**

where

|               |                                                                                                                                                                                                     |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>number</b> | is the line number of the statement                                                                                                                                                                 |
| <b>var</b>    | is a simple (nonsubscripted) variable, identical in both the FOR and NEXT statements of the couplet                                                                                                 |
| <b>expi</b>   | is a number or expression whose value is the initial value of <b>var</b>                                                                                                                            |
| <b>expf</b>   | is a number or expression whose value is the final value of <b>var</b>                                                                                                                              |
| <b>exps</b>   | is a number or expression whose value is the increment between successive values of <b>var</b> between <b>expi</b> and <b>expf</b> (if omitted from the statement, <b>exps</b> is assumed to be +1) |

The FOR statement enters a loop and the NEXT statement exits from the loop, directing the computer back to the FOR statement until **expf** is reached. At this point the NEXT statement allows the computer to exit from the loop and pass to the next statement in the program.

Specifications and restrictions in the use of loops, as well as examples of FOR and NEXT statements, are given in the discussion of loops (section 1.3).

## IF/THEN Statement

This statement has the format

**number**      **IF** **exp** **rel** **exp** **THEN** **next**

where

|               |                                             |
|---------------|---------------------------------------------|
| <b>number</b> | is the line number of the IF/THEN statement |
| <b>exp</b>    | is a mathematical expression or number      |
| <b>rel</b>    | is an arithmetic operation or relationship  |

**exp** is a mathematical expression or number used if and only if *rel* is present

**next** is the line number of the statement to be executed next if the preceding conditions are met

The conditions of an IF/THEN statement are met when the logical argument **exp rel exp** is true or if the single mathematical expression **exp** is nonzero. Any expression can be considered a logical argument by evaluating the numbers represented in it by the logical constants false = 0 and true = not 0.

#### Examples:

```
40    IF SIN (X) = M THEN 80
```

where the computer jumps to line 80 if the sine of x is less than or equal to m, but otherwise goes to the next line after 40.

```
20    IF G = 0 THEN 65
```

where the computer jumps to line 65 if G is zero, but otherwise goes to the next line after 20.

```
35    IF A THEN 83
```

where the computer jumps to line 83 if A is nonzero, but otherwise goes to the next line after 35.

```
85    IF A + B - 5 THEN 302
```

where the computer jumps to line 302 if the value of the expression is nonzero, but otherwise goes to the next line after 85.

```
90    IF -2 THEN 200
```

where the computer always jumps to line 200 since the value of the expression is always nonzero, i.e., such a statement is equivalent to an unconditional GO TO statement (section 3.7).

## GO TO Statements

There are two types of GO TO statements. The **unconditional GO TO statement** causes the program execution to jump to the specified line every time the statement is encountered. This statement has the format

```
number    GO TO next
```

## statements in basic

where

|               |                                                         |
|---------------|---------------------------------------------------------|
| <b>number</b> | is the line number of the GO TO statement               |
| <b>next</b>   | is the line number of the statement to be executed next |

The **computed GO TO statement** specifies several lines for the jump. The one selected depends on the value *n* of an expression in the statement, where the line number chosen is the *n*th line number in the statement. This statement has the format

**number          GO TO exp OF next,next,next,...**

where

|               |                                                                                                                     |
|---------------|---------------------------------------------------------------------------------------------------------------------|
| <b>number</b> | is the line number of the GO TO statement                                                                           |
| <b>exp</b>    | is an expression                                                                                                    |
| <b>next</b>   | is a line number of one of the statements that,<br>depending on the value of <b>exp</b> , is to be<br>executed next |

Thus, **exp** is evaluated and the answer truncated to the integer value *n* that determines the **next** to be used, e.g., if *n* = 2, the second **next** is the valid line number.

The IF/THEN statement (section 3.6) is also called the **conditional GO TO statement** because the program jumps to the specified line only if a certain relationship exists.

**Examples:** *Unconditional GO TO Statement:*

150 GO TO 75

where the next statement to be executed is the one in line 75.

*Computed GO TO Statement:*

160 GO TO I-3 OF 10,30,100

where the next statement to be executed is the one in line 30 when *I* = 5, since *I*-3 = 2, selecting the second line number in the series.

*Conditional GO TO Statement:* See IF/THEN statement (section 3.6).

## GOSUB, RETURN, and SUB Statements

There are three types of GOSUB (Go To Subroutine) statements. All of them direct the computer to a subroutine according to the specifications of the statement, and all of them are used with the **RETURN statement**, which has the format

**number          RETURN**

where **number** is the line number of the RETURN statement. The RETURN statement is

the exit from the subroutine and returns the computer to the first line number greater than that of the calling GOSUB statement. RETURN is the only exit from a subroutine, i.e., GO TO or IF/THEN statements cannot be used to exit from a subroutine. There can be more than one RETURN statement in a subroutine, but only one of them can be used on any given pass through the subroutine.

Subroutines can be nested, i.e., GOSUB statements can be used inside subroutines to call sub-subroutines.

The **unconditional GOSUB statement** causes the program execution to jump to the specified line every time the statement is encountered. This statement has the format

**number            GOSUB subr**

where

|               |                                                                                 |
|---------------|---------------------------------------------------------------------------------|
| <b>number</b> | is the line number of the GOSUB statement                                       |
| <b>subr</b>   | is the line number of the first statement in the subroutine to be executed next |

The **computed GOSUB statement** specifies several lines for the subroutine jump. The one selected depends on the value *n* of an expression in the statement, where the line number chosen is the *n*th line number in the statement. This statement has the format

**number            GOSUB exp OF subr,subr,subr,...**

where

|               |                                                                                                                                                       |
|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>number</b> | is the line number of the GOSUB statement                                                                                                             |
| <b>exp</b>    | is an expression                                                                                                                                      |
| <b>subr</b>   | is the line number of one of the statements that, depending on the value of <b>exp</b> , is the first statement of the subroutine to be executed next |

Thus, **exp** is evaluated and the answer truncated to the integer value *n* that determines the **subr** that will be used, e.g., if *n* = 2, the second **subr** is the valid line number of the first statement in the subroutine selected.

The **GOSUB statement with parameters** enters parameter values in the subroutine specified. The statement has the format

**number            GOSUB subr,param,param,...**

where

|               |                                                                                                                  |
|---------------|------------------------------------------------------------------------------------------------------------------|
| <b>number</b> | is the line number of the GOSUB statement                                                                        |
| <b>subr</b>   | is the line number of the first statement in the subroutine to be executed next, i.e., that of the SUB statement |
| <b>param</b>  | is a parameter value to be entered into the subroutine by the SUB statement                                      |

## statements in basic

The GOSUB statement with parameters is always used with a **SUB statement** that has the format

**subr**            **SUB** var,var,...

where

**subr**            is the line number of the SUB statement and is  
                 identical with the **subr** in the corresponding  
                 GOSUB statement

**var**            is a variable

The parameter values in the GOSUB statement are assigned to the corresponding variables in the SUB statement; i.e., the first parameter value is assigned to the first variable, etc. The effect of a SUB statement or the definition of variables is removed upon execution of the corresponding RETURN statement. A subroutine defined by a SUB statement can be entered retrogressively (see example below).

**Examples: Unconditional GOSUB Statement:** This program for finding the greatest common denominator (GCD) of three integers using the Euclidean algorithm illustrates the use of the unconditional GOSUB statement. The first two numbers are selected in lines 30 and 40 and their GCD is determined in the subroutine, lines 200 through 310. This GCD is called X in line 60, the third number is called Y in line 70, and the subroutine is entered from line 80 to find the GCD of these two numbers. The resulting GCD is the greatest common divisor of the three given numbers and is printed with them in line 90.

### Program

```
READY
10  PRINT "A", "B", "C", "GCD"
20  READ A, B, C
30  LET X = A
40  LET Y = B
50  GOSUB 200
60  LET X = G
70  LET Y = C
80  GOSUB 200
90  PRINT A, B, C, G
100 GO TO 20
110 DATA 60, 90, 120
120 DATA 38456, 64872, 98765
130 DATA 32, 384, 72
200 LET Q = INT(X/Y)
210 LET R = X - Q*Y
220 IF R = 0 THEN 300
230 LET X = Y
240 LET Y = R
250 GO TO 200
300 LET G = Y
310 RETURN
320 END
RUN
```

| Result              |       |       |     |
|---------------------|-------|-------|-----|
| A                   | B     | C     | GCD |
| 60                  | 90    | 120   | 30  |
| 38456               | 64872 | 98764 | 4   |
| 32                  | 384   | 72    | 8   |
| ERROR 56 IN LINE 20 |       |       |     |

*Computed GOSUB Statement:* The expression in the statement is evaluated and truncated to the integer n, and program execution jumps to the nth line number in the line number list. If, when  $z = 1.68$ , we want the program execution to jump to line 55, we can write

```
20  GOSUB Z+1 OF 30,55,70
```

where the expression is computed to be equal to 2.68 and is truncated to the integer 2, thus selecting the second line number. In this case, the computed GOSUB results in a jump like that obtained with the unconditional GOSUB

```
20  GOSUB 55
```

*GOSUB Statement with Parameters:* To pass the two parameter values 5 and 10 into a subroutine defined by a SUB statement in line 100, we can write

```
20  GOSUB 100, 5, 10
```

or, the same transfer results when we use a variable j when its value is 6 and we write

```
20  GOSUB 100, 5, J+4
```

*GOSUB Statement with Parameters and SUB Statement for Retrogressive Subroutine Entry:* This program shows the use of these statements for a simple incrementation.

#### Program

```
LIST
10  REM SHOW USE OF GOSUB WITH PARAMETER TRANSFER
20  LET N = 5
30  GOSUB 100, N
40  PRINT "MAIN PROGRAM"; N
50  END
100 SUB N
105 REM THIS EXAMPLE USES A RETROGRESSIVE SUBR. ENTRY
110 PRINT "SUBROUTINE GOT"; N
120 IF N = 0 THEN 140
130 GOSUB 100, N-1
140 RETURN
9999 END
RUN
```



**Result**

```

SUBROUTINE GOT 5
SUBROUTINE GOT 4
SUBROUTINE GOT 3
SUBROUTINE GOT 2
SUBROUTINE GOT 1
SUBROUTINE GOT 0
MAIN PROGRAM 5
READY

```

**PRINT Statements**

The PRINT statements control the output and format of the results of BASIC programs.

**General Types**

There are four common uses of PRINT statement:

- To print the results of computations
- To print comments
- To print a combination of the above
- To skip a line

Each of these uses requires a particular format of PRINT statement.

To print the results of computations, use the format

**number      PRINT exp,exp,...**

where

**number**      is the line number of the statement  
**exp**          is a variable or expression whose computed value  
                  is to be printed (up to 5 values per line)

The variables used must already have been given values.

*Examples:* To print the value of x and the value of its square root, we can write

```
100 PRINT X, SQR(X)
```

To print the values of the five expressions x, y, z,  $b^2 - 4ac$ , and e to the power a - b, we can write

```
135 PRINT X, Y, Z, B*B-4*A*C, EXP(A-B)
```

To print comments, use the format

```
number      PRINT "comment"
```

where

**number** is the line number of the statement  
**comment** is the material to be printed

Within **comment** blanks will be observed by the computer. If several comments are to be printed on one line, e.g., for column headings, use the format

```
number      PRINT "comment", "comment", ...
```

where each **comment** is enclosed within quotation marks. In no case are the quotation marks printed. When used for column headings, the printout automatically aligns the headings and the columns because the commas specify that the next heading be printed in the next zone (section 3.9.2).

*Examples:*

```
100  PRINT "NO UNIQUE SOLUTION"
430  PRINT "X VALUE", "SIN", "RESOLUTION"
```

To print a combination of results and comments, use the PRINT statement with the comments in quotation marks, and with the variables or expressions whose values are to be printed given as above.

*Examples:* Where  $x = 625$ , we can write

```
15   PRINT "THE VALUE OF X IS" X
30   PRINT "THE SQUARE ROOT OF" X; "IS" SQR(X)
```

and obtain the printout

```
THE VALUE OF X IS 625
THE SQUARE ROOT OF 625 IS 25
```

Note that no terminator (semicolon or comma) is required after quotation marks, but they are required after variables or expressions (except as the final item in the statement). The comma causes the item that follows to be printed in the next zone, while a semicolon causes it to be printed closed-up (section 3.9.2).

To skip a line, use the format

```
number      PRINT
```

where **number** is the line number of the statement.

statements in basic

This statement simply requests that the computer print nothing and then activate the carriage return, i.e., skip a line.

Manipulating the Printing Format

The Teletype line is divided into five **printing zones**, starting at positions 0, 15, 30, 45, and 60, respectively. A **terminator** (comma or semicolon) controls the use of these zones. A **comma** (,) moves printing to the next printing zone, or, if the fifth printing zone has been filled, to the first printing zone of the next line. A **semicolon** (;) produces more compact output since it inhibits spacing between printing zones, acting only to separate quantities to be printed (e.g., A + B;C/D), or to suppress a carriage return at the end of a print statement.

Spacing within a printing zone depends on the value and type of the number being printed. A number is always printed in a zone larger than it needs, and is left-justified in that zone. The zone size is determined as follows:

| Value of Number                                            | Type of Number                           | Format of Zone                                                                                                                                                                              |
|------------------------------------------------------------|------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $-999 \leq n \leq +999$                                    | Integer                                  | $\sqrt{\sqrt{\hspace{1cm}}}$                                                                                                                                                                |
| $-999999 \leq n \leq -1000$<br>$+1000 \leq n \leq +999999$ | Integer                                  | $\wedge \hspace{.5cm} \wedge \wedge \wedge$<br>$-99999 \leq n \leq -1000 \text{ INT}$<br>$+1000 \leq n \leq 99999$                                                                          |
| $0.1 \leq n \leq 999999.5$                                 | Real (normal range)                      | $\sqrt{\sqrt{\sqrt{\hspace{1cm}}}}$<br>$-999999 \leq n \leq -10000 \text{ INT}$<br>$+10000 \leq n \leq 999999$<br>(decimal point<br>printed as one of<br>x's; trailing zeros<br>suppressed) |
| $n < 0.1$<br>$999999.5 < n$                                | Large integer<br>or real (extreme range) | $\simeq \text{x.xxxxxx E } \pm \text{ee } \wedge \wedge \wedge$                                                                                                                             |

The carat (^) represents a space typed on the Teletype.

For example, for the program

```
10  FOR I = 1 TO 15
20  PRINT I
30  NEXT I
40  END
RUN
```

the Teletype prints 1 at the beginning of a line, 2 at the beginning of the next line, etc., up to 15 on the 15th line.

By changing line 20 to read

```
20 PRINT I,
RUN
```

the numbers are printed in zones, reading

|    |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 |

To print the numbers in more tightly packed zones, replace the comma in line 20 with a semicolon

```
20 PRINT I;
RUN
```

and the result is

|    |    |    |    |    |   |   |   |   |    |
|----|----|----|----|----|---|---|---|---|----|
| 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |   |   |   |   |    |

A character string in quotation marks is printed just as it appears. The end of a PRINT line always signals a new line unless a comma or a semicolon is the last symbol. Thus, the statement

```
50 PRINT X, Y
```

prints the two numbers and returns to the next line, while the statement

```
50 PRINT X,Y,
```

prints these two values but does not return. The next number is printed in the third zone, following the values of X and Y in the first two zones.

Since the end of a PRINT statement signals a new line,

```
250 PRINT
```

skips one line. This can be used to put a blank line in the program to allow vertical spacing of the results. It can also be used to complete a partially filled line:

```
50 FOR M = 1 TO N
110 FOR J = 1 TO M+1
```

```

120 PRINT B(M,J);
130 NEXT J
140 PRINT
150 NEXT M

```

This program prints B(1,1) followed by B(1,2). Without line 140, the Teletype would continue to print B(2,1), B(2,2), and B(2,3) on the same line, and then B(3,1), B(3,2), etc. Line 140 directs the Teletype to start a new line after printing the B(1,2) value corresponding to M = 1, and again after printing the value of B(2,3) corresponding to M = 2, etc.

The following rules for the printing of numbers will aid in interpreting printed results:

- If the number is an integer with a value from -999999 to +999999, inclusive, the decimal point is not printed.
- If the number is real and has an absolute value between 0.1 and 999999.5, it is rounded to six digits and printed with a decimal point. Trailing zeros after the decimal point are suppressed.
- A number either greater than 999999.5 or less than 0.1 is rounded to six places. The Teletype then prints a space (if positive) or a minus sign (if negative), the first digit, the decimal point, the next five digits, the letter E (exponent), the sign of the exponent, and the value of the exponent. For example, 3.243756 is printed as 3.243756E+6.

The following program to print the powers-of-two shows how numbers are printed.

### Program

```

READY
10 FOR N = -5 TO 30
20 PRINT 2^N;
30 NEXT N
40 END
RUN

```

### Result

```

3.12501E-02   6.25002E-02   .125   .25   .5   1
1.99999   3.99999   7.99997   15.9999   31.9998   63.9995
127.999   255.998   511.994   1023.99   2047.98   4095.94
8191.88   16383.7   32767.4   65535   131069   262138
524277   1.04855E+06   2.09790E+06   4.19422E+06   8.38839E+06
1.67768E+07   3.35536E+07   6.71068E+07   1.34213E+08   2.68427E+08
5.36854E+08   1.07370E+09   READY

```

The instructions

```
50  PRINT "VARIAN BASIC ";
51  PRINT "LANGUAGE COMPILER"
```

print

```
VARIAN BASIC LANGUAGE COMPILER
```

Output formatting can be controlled even further by use of the function TAB. Insertion of TAB(17) causes the Teletype to move to column 17 as if a tab had been set there. For this purpose, line positions are numbered 0 through 71.

TAB can contain any expression as its argument. The value of the expression is computed, truncated, and its integer part taken. The Teletype then moves forward to this position. If the position has already been passed, the TAB is ignored. If the result is greater than 71, the Teletype moves to position 0 of the next line.

For example, to insert the following line in a loop, we write

```
PRINT X; TAB(12); Y; TAB(27); Z
```

This prints the X value in column 0, the Y value in column 12, and the Z value in column 27.

A comma following a TAB clause has no effect on Teletype positioning. The statement

```
PRINT TAB(7), A+B
```

prints the value of A + B starting at position 7. The statement

```
PRINT Z, A+B
```

prints the value of A + B at position 15 (second printing zone).

## INPUT Statement

The INPUT statement has the format

```
number    INPUT var,var,...
```

where

|               |                                     |
|---------------|-------------------------------------|
| <b>number</b> | is the line number of the statement |
| <b>var</b>    | is a variable                       |

The INPUT statement acts as a READ statement but does not draw data from DATA

## statements in basic

statements. Instead, it asks the user to supply the required data by outputting a question mark. The user types the data, separating each number from the next with a comma, and presses the RETURN key on the Teletype.

INPUT should be used only when small amounts of data are to be entered, or when it is necessary to enter data during the running of the program, since data entry via INPUT is slow. Furthermore, *data entered via INPUT statements are not saved with the program.* Numbers input should not exceed 9 digits.

To retake control from the INPUT processor, type a plus (+).

**Examples:** If the user is to supply values for x and y, type

```
40 INPUT X, Y
```

before the first statement that uses either variable. When the computer encounters this statement, it types a question mark. Type two numbers, separated by a comma and press the RETURN key. The computer then goes on with the rest of the program.

Frequently, an INPUT statement is combined with a PRINT statement to ensure that the user knows what the question mark is asking for. You might type, for example,

```
20 PRINT "YOUR VALUES OF X, Y, AND Z ARE";  
30 INPUT X, Y, Z
```

and the computer will type

```
YOUR VALUES OF X, Y, AND Z ARE?
```

(Without the semicolon at the end of line 20, the question mark would have been printed on the next line.)

## RESTORE Statement

The RESTORE statement has the format

```
number RESTORE
```

where **number** is the line number of the statement.

The RESTORE statement permits the reuse of data within a program. When RESTORE is encountered in a program, the computer restores the data-block pointer to the first data number. A subsequent READ statement then starts the reading of the data again from the first data item.

If the desired data items are preceded by unwanted data, use extra READ statements to pass over these numbers.

**Example:** This program reads the data, restores the data-block pointer, and rereads the data. Note the use of line 570 to pass over the already-known value of n.

```

100  READ N
110  FOR I = 1 TO N
120  READ X
    .
    .
    .
200  NEXT I
    .
    .
    .
560  RESTORE
570  READ X
580  FOR I = 1 TO N
590  READ X

```

## REM (Remark) Statement

The REM statement has the format

**number        REM comment**

where

**number**        is the line number of the statement  
**comment**       is any comment desired in the listing

The comment in the statement is printed in the listing just as written, including blanks. It is, however, otherwise ignored by the processor.

The line number of a REM statement can be used in a GO TO or IF/THEN statement.

**Examples:**

```

100  REM INSERT DATA IN LINE 900-998.  THE FIRST
110  REM NUMBER IS N, THE NUMBER OF POINTS.  THEN
120  REM THE DATA POINTS THEMSELVES ARE ENTERED, BY

200  REM THIS IS A SUBROUTINE FOR SOLVING EQUATIONS
    .
    .
    .
300  RETURN
    .
    .

```



520 GOSUB 200

## CALL Statement

The CALL statement has the format

**number**      **CALL** *asubr,parameter,parameter,...*

where

**number**      is the line number of the statement  
**asubr**        is the name of an assembly-language subroutine  
                   (1-6 alphanumeric characters)  
**parameter**    is a variable, number, or expression

The CALL statement links absolute assembly-language subroutines to BASIC. Execution of the CALL statement passes control to an assembly-language program appended to the BASIC system. For internal design considerations of CALLED subroutines, see appendix B.

Results can be returned to the BASIC program through the *parameters*. *Care is necessary to prevent an constant from being placed in a position in the CALL statement where the subroutine attempts to return a value. This results in the value of the constant being changed throughout the BASIC interpreter.*

**Examples:**

```
100 CALL SUBA, A/D, X, 3
200 CALL RESET
883 CALL POLY, A+3, B+4, C*D+5, E
```

## WAIT Statement

The WAIT statement has the format

**number**      **WAIT** *delay*

where

**number**      is the line number of the statement  
**delay**        is a number, variable, or expression whose value  
                   gives the number of milliseconds delay introduced  
                   into the program at this point (maximum value  
                   32,767)

## STOP Statement

The STOP statement has the format

**number**      **STOP**

where **number** is the line number of the STOP statement.

The STOP statement stops the computer execution of the program. Execution resumes with the next BASIC statement when the RUN or START key on the computer is pressed.

## END Statement

The END statement has the format

**number END**

where **number** is the line number of the END statement.

The END statement returns control to the operator. A program can have more than one END statement, but it is possible for a program to terminate without reaching an END statement, e.g., when there is no more data to process.



## SECTION 4 – USING THE BASIC SYSTEM

This section provides:

- The operating instructions for the BASIC system
- The control commands that the user inputs from the Teletype keyboard
- An explanation of the two BASIC operating modes: program mode and calculator mode

### OPERATING INSTRUCTIONS

To operate the BASIC system:

- a. Manually enter the bootstrap loader program (refer to the applicable system reference handbook).
- b. Load the BASIC system tape with the Binary Load/Dump program provided.
- c. Start the program at location 02. The program types:

**PAPER TAPE: TYPE 1 FOR HI SPEED, ELSE TTY**

- d. If the high-speed paper tape reader/punch is desired, enter a 1. Any other entry assigns paper tape input/output to the Teletype. The program types:

**MAT, TRIG: 1 TO SAVE BOTH, 2 TO SAVE TRIG, ELSE  
DELETE BOTH**

- e. The lower boundary of the BASIC table space is selected here. The program space used by the matrix or trig function can be deleted and used by BASIC for working storage. If there is more than 8K or memory, the program types:

**AID: 1 TO SAVE, ELSE WIPEOUT**

The core space occupied by the utility program AID will be used by BASIC unless there is a request that it be saved. In an 8K system, AID is always lost. The program types:

### **READY**

and waits for input from the Teletype.

- f. Start the input, which can be either a BASIC statement or a control command (section 4.2).

If, in the process of typing a statement, you make a typing error and notice it immediately, you can correct it by pressing the backward arrow (shift key above the letter O). This deletes the character in the preceding space, and you can then type in the correct character. Pressing this key a number of times will erase from a line the characters in that number of preceding spaces. To delete all of the present line, press RUBOUT. Programs or data can be annotated by typing the remark and then deleting the line (as far as the system is concerned) with a RUBOUT. BASIC types a backslash to show that a line has been deleted.

After typing your complete program, type RUN, press the RETURN key. If the program is free of errors, the computer will run it and type out any results that you requested in your PRINT statements. This does not mean that your program is entirely correct, but that it has no errors of the type known as grammatical errors.

If there are grammatical errors, the computer types an error code as soon as each error is detected while the program is being typed. Errors detected after RUN are structural (loop-nesting, matching GOSUB and return) or arithmetic errors. A list of error codes with an interpretation of each is in appendix A.

If you receive an error message, correct the error by typing a new line with the correct statement. For example, to eliminate the statement on line 110 from a program, type 110 and then press the carriage return. To insert a statement between those on lines 60 and 70, give it a line number between 60 and 70.

If it is obvious while the computer is running that the answers are wrong, press any Teletype key and computation ceases. The system types:

### **READY**

and you can make your corrections.

**Example:****Program**

```

READY
10  FOR N = 1 TO 7
20  PRINT N, SQR(N)
30  NEXT N
40  PRINT "DONE"
50  END
RUN

```

**Result**

```

1      1
2      1.41421
3      1.73205
4      2
5      2.23607
6      2.44948
7      2.64575
DONE

```

*At all times, there is only one device from which the BASIC interpreter will accept information and only one device on which it will output data. These devices are fixed at program initialization time (section 3.1).*

**CONTROL COMMANDS**

The following are BASIC control commands:

|                  |                                                                                                                                                          |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Any key</b>   | Stops program execution when the program is running. Returns control to the Teletype.                                                                    |
| <b>RUN nnn</b>   | Begins execution of the program starting at line nnn. If no digits (nnn) are entered, 1 is assumed.                                                      |
| <b>LIST nnn</b>  | Outputs an up-to-date listing of the program on the currently active listing device starting at line nnn. Listing can be terminated by pressing any key. |
| <b>PLIST nnn</b> | Same as <b>LIST</b> , but the output device is the high-speed paper tape punch or Teletype, whichever was selected.                                      |

**PTAPE** Moves control to the high-speed paper tape reader or Teletype, whichever was selected.

**SCRATCH** Deletes the current BASIC program from memory.

## PROGRAM AND CALCULATOR MODES

The description of the BASIC system to this point has assumed that the definition and execution of programs has occurred at different times, i.e., that one entered a set of numbered BASIC statements that were checked for validity and stored, then entered the control command RUN to start execution of the program defined by this set of statements. This mode of operation is called the **program mode**.

Varian BASIC can also run in the **calculator mode**, which causes the computer to respond immediately to a BASIC statement, just like a desk calculator. In this mode, one does not enter a statement number, but only a statement, and the BASIC system executes it immediately. *Example:*

```
PRINT 5*4-3
```

```
17
```

## APPENDIX A — ERROR MESSAGES

An error message is printed as soon as the error condition is detected. The format is as follows:

| ERROR xx IN LINE nn |                                                                   |
|---------------------|-------------------------------------------------------------------|
| Error Code xx       | Meaning                                                           |
| 0                   | Hardware M/D option missing                                       |
| 1                   | Statement ends unexpectedly                                       |
| 2                   | Input exceeds 72 characters                                       |
| 3                   | System command not recognized (can be missing statement number)   |
| 4                   | Missing or incorrect statement type                               |
| 5                   | Exponent of number is missing power                               |
| 6                   | Symbol following MAT not recognized                               |
| 7                   | LET statement has no store                                        |
| 9                   | Missing or incorrect function identifier in DEF                   |
| 10                  | Missing parameter in DEF statement                                |
| 11                  | Missing assignment operator                                       |
| 12                  | Missing THEN                                                      |
| 13                  | Missing or incorrect FOR variable                                 |
| 14                  | Missing TO                                                        |
| 15                  | Incorrect STEP in FOR statement                                   |
| 16                  | Called routine does not exist                                     |
| 17                  | Wrong number of parameters in CALL statement                      |
| 18                  | Missing or incorrect constant in DATA statement                   |
| 19                  | Missing or incorrect variable in READ statement                   |
| 20                  | No closing quotation mark for PRINT string                        |
| 21                  | Missing print delimiter or bad PRINT quantity                     |
| 22                  | Illegal word follows MAT                                          |
| 23                  | Missing delimiter                                                 |
| 24                  | Improper matrix function                                          |
| 25                  | No subscript where expected                                       |
| 26                  | Cannot invert or transpose matrix into itself                     |
| 27                  | Missing multiplication operator                                   |
| 28                  | Improper matrix operator                                          |
| 29                  | Matrix cannot be both operand and result of matrix multiplication |
| 30                  | Missing left parenthesis                                          |



| Error Code xx | Meaning                                                                   |
|---------------|---------------------------------------------------------------------------|
| 31            | Missing right parenthesis                                                 |
| 32            | Operand not recognized                                                    |
| 33            | Defined array missing subscript part                                      |
| 34            | Missing array identifier                                                  |
| 35            | Missing or bad integer                                                    |
| 36            | Nonblank characters following statement's logical end                     |
| 37            | Out of storage during syntax phase                                        |
| 38            | Paper tape reader not ready or EOF on paper tape                          |
| 39            | Doubly defined function                                                   |
| 40            | FOR statement has no matching NEXT statement                              |
| 41            | NEXT statement has no matching FOR statement                              |
| 42            | Formal parameter finds no actual parameter                                |
| 43            | Array appears with inconsistent dimensions                                |
| 44            | Missing END statement or attempted execution of a nonexecutable statement |
| 45            | Array doubly dimensioned                                                  |
| 46            | Number of dimensions not obvious                                          |
| 47            | Array too large                                                           |
| 48            | Out of storage during array allocation                                    |
| 49            | Subscript too large                                                       |
| 50            | Accessed operand has undefined value                                      |
| 51            | Noninteger power of negative number                                       |
| 53            | Missing statement                                                         |
| 55            | RETURN finds no address                                                   |
| 56            | Out of data                                                               |
| 57            | Out of storage during execution                                           |
| 58            | Dynamic array exceeds allocated storage                                   |
| 59            | Dimensions not compatible                                                 |
| 60            | Matrix operand contains undefined element                                 |
| 61            | Singular or nearly singular matrix                                        |
| 62            | Trigonometric function argument too large                                 |
| 63            | Attempted square root of negative argument                                |
| 64            | Attempted log of negative argument                                        |

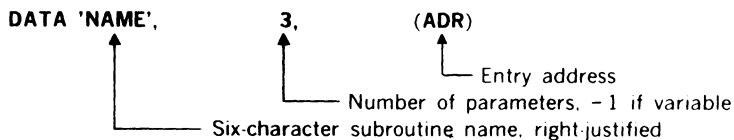
The following errors are warnings only; program execution continues:

|    |                                                             |
|----|-------------------------------------------------------------|
| 65 | Numerical overflow; result taken to be $\pm$ infinity       |
| 66 | Numerical underflow; result taken to be zero                |
| 67 | Log of zero taken to be $-\infty$                           |
| 68 | EXP overflow; result taken to be $+\infty$                  |
| 69 | Division by zero; result taken to be $\pm$ infinity         |
| 70 | Zero raised to negative power; result taken to be $+\infty$ |

## APPENDIX B – CALL DESIGN CONSIDERATIONS

There are certain internal design considerations for CALL subroutines. Parameters are passed to the called routine in the convention adopted by Varian; i.e., a list of the addresses of the actual parameters immediately following the JMPM instruction to the subroutine. All actual parameters are real (Varian floating-point format) numbers.

Subroutine linkage to the BASIC interpreter is through a list containing the name of the subroutine, the number of parameters it expects, and its entry address. The linkage list contains five words per subroutine, construction as follows:



The setting of the first word of the name field to zero signals the end of the list. A pointer to the linkage list must be placed in location 010. Location 022 points to the highest address used by BASIC, so this cell must be decremented to allow space for the linkage list and the user subroutines to be loaded above the BASIC tables.

A set of address bounds are available if the programmer wishes to know what type of parameter has been passed to him. By checking the parameters against these bounds, he can ensure, for example, that he does not store into constants. These address bounds are stored in the following location pairs (the first cell holds the first address of the block; the second holds the last address + 1):

|                 |                         |
|-----------------|-------------------------|
| Simple variable | $(011) \leq X < (012)$  |
| Expression      | $(015) \leq X < (0117)$ |
| Constant        | $(017) \leq X < (020)$  |

The following program shows a setup to make the two assembly language subroutines SUB1 and SUB2 available to a basic program.

**To operate this program:**

- Load and initialize BASIC, selecting options as requested, until **READY** is typed.
- Load the subroutine package containing SUB1 and SUB2 using the Binary Load/Dump program and overlaying locations 010 and 022.
- Restart BASIC at location 02 to access the two subroutines.

SYMBOLS

1 015006 R TABL  
1 015003 R SUB2  
1 015000 R SUB1  
1 015000 LOC

\*  
\*DEMONSTRATION OF TYPICAL SUBROUTINE ADDITION TO BASIC  
\*

015000 015000 LOC ,EQU ,015000  
015000 000000 SUB1 ,ORG ,LOC  
\* ,ENTR , LOC OF SUBROUTINES  
\* BODY OF SUB1 ENTRY TO SUB1

015001 001000 ,JMP\* ,SUB1 RETURN FROM SUB1  
015002 115000 R  
015003 000000 SUB2 ,ENTR , ENTRY TO SUB2  
\* BODY OF SUB2  
\* RETURN FROM SUB2

015004 001000 ,JMP\* ,SUB2  
015005 115003 R  
015006 015006 R TABLE ,EQU ,\* LOC OF LINKAGE TABLE  
015006 120240 ,DATA , SUB1' ,0,(SUB1) SUB1 ENTRY, ZERO PARA  
015007 151725  
015010 141261

```
015011      000000
015012      015000 R
015013      120322
015014      142723
015015      142724
015016      000002
015017      015003 R
015020      000000

          ,DATA      , '      RESET' , 2, (SUB2) SUB2 ENTRY, TWO PARA

          ,DATA      , 0      **END FLAG FOR LINK TABLE**

*
*NOW SET POINTERS FOR INTERPRETER
*
000010      000010      ,ORG      , 010      LOC OF INTERPRETER POINTERS
000010      015006 R      ,PZE      , (TABL)      POINTER TO LINKAGE TABLE
000022      ,ORG      , 022
000022      014777      ,PZE      , (LOC-1)      POINTER TO LAST CELL
          000002      ,END      , 2

LITERALS

POINTERS

SYMBOLS

1      015006 R      TABL
1      015003 R      SUB2
1      015000 R      SUB1
1      015000      LOC
```



# **Report Program Generator IV (RPG IV) Reference Manual**





## TABLE OF CONTENTS

### SECTION 1 INTRODUCTION

|                                     |      |
|-------------------------------------|------|
| STRUCTURE OF RPG IV PROGRAMS.....   | I-1  |
| STATEMENTS .....                    | I-3  |
| Data-Defining Statements.....       | I-3  |
| Procedural Statements.....          | I-4  |
| ELEMENTS OF RPG IV STATEMENTS ..... | I-5  |
| Statement Numbers.....              | I-5  |
| Names.....                          | I-6  |
| Conditions.....                     | I-7  |
| Indicators.....                     | I-8  |
| Constants.....                      | I-9  |
| Literals .....                      | I-9  |
| Expressions .....                   | I-10 |
| Comment Lines .....                 | I-12 |
| SAMPLE PROGRAMS .....               | I-13 |

### SECTION 2 SYSTEM CONCEPTS

|                |      |
|----------------|------|
| HARDWARE ..... | II-1 |
| SOFTWARE.....  | II-2 |



### SECTION 3 RPG IV STATEMENTS

|                                 |            |
|---------------------------------|------------|
| DATA-DEFINING STATEMENTS.....   | III-2      |
| Record Statement .....          | III-2      |
| Record Field Statement.....     | III-4      |
| Table Statement.....            | III-11     |
| Table Field Statement .....     | III-13     |
| <br>PROCEDURAL STATEMENTS ..... | <br>III-15 |
| PROCEDURE Statement .....       | III-16     |
| MOVE Statement.....             | III-16     |
| MOVEZ STATEMENT.....            | III-17     |
| POST Statement.....             | III-17     |
| COLLECT Statement.....          | III-18     |
| ADD Statement .....             | III-19     |
| COMPUTE Statement.....          | III-19     |
| GO TO Statement .....           | III-20     |
| Indexed GO TO Statement.....    | III-20     |
| PERFORM Statement.....          | III-21     |
| RETURN Statement .....          | III-22     |
| SET Statement .....             | III-23     |
| ENTER Statement .....           | III-23     |
| DELETE Statement.....           | III-24     |
| LOOKUP Statement.....           | III-25     |
| CALL Statement .....            | III-26     |
| READ CARD Statement.....        | III-27     |
| PRINT Statement.....            | III-27     |
| PUNCH Statement .....           | III-29     |
| STOP Statement.....             | III-29     |
| END Statement .....             | III-29     |

**SECTION 4**  
**OPERATING PROCEDURES**

|                                                         |      |
|---------------------------------------------------------|------|
| COMPILING AN RPG IV PROGRAM .....                       | IV-2 |
| Preparation of the Deck for Compilation .....           | IV-2 |
| Operating the Hardware for RPG IV Compilation .....     | IV-3 |
| Compilation Errors .....                                | IV-5 |
| PROCESSING DATA WITH A COMPILED RPG IV PROGRAM .....    | IV-8 |
| Preparation of the Deck for Processing Data .....       | IV-8 |
| Operating the Hardware for Processing RPG IV Data ..... | IV-8 |

**SECTION 5**  
**SAMPLE COMPILATION LISTING**

**APPENDICES**

**APPENDIX A**  
**INDICATOR CHART**

**APPENDIX B**  
**COLLATING SEQUENCE AND CODES**

**APPENDIX C**  
**COMPILATION ERROR MESSAGES**

**APPENDIX D**  
**CARD BOOTSTRAP LOADER**

**APPENDIX E**  
**CALL STATEMENT SUBROUTINE USAGE**

## LIST OF ILLUSTRATIONS

|      |                                              |      |
|------|----------------------------------------------|------|
| I-1  | Layout of a Typical RPG IV Program.....      | I-2  |
| I-2  | Flowchart for the Sample RPG IV Program..... | I-14 |
| I-3  | Sample RPG IV Program.....                   | I-15 |
| I-4  | Data for Sample RPG IV Program.....          | I-16 |
| I-5  | Final Report from Sample RPG IV Program..... | I-16 |
| IV-1 | Deck for Compiling an RPG IV Program.....    | IV-4 |
| IV-2 | Runtime Deck for Processing RPG IV Data..... | IV-9 |
| V-1  | Calendar Program.....                        | IV-8 |
| V-2  | Flowchart for Calendar Program.....          | V-3  |
| V-3  | Input Data.....                              | V-7  |
| V-4  | Printed Output.....                          | V-8  |

## **FOREWORD**

### **RPG IV: DEFINITION AND PURPOSE**

The *RPG (Report Program Generator) IV language* is an advanced version of the widely used RPG commercial and general data-processing systems. RPG IV permits the concise coding of powerful programs in a simple and efficient manner. Thus, users with backgrounds other than data processing can use RPG IV problem-solving techniques without extensive training or practice.

RPG IV improves on basic RPG in that it incorporates many automatic features and powerful procedural statements. RPG IV is particularly adapted to processing data for the output of reports, but has many other applications as well.

### **ORGANIZATION OF THE MANUAL**

This manual is divided into a foreword plus five chapters.

The foreword defines RPG and explains its purpose. It also explains the layout of the manual and gives a bibliography of prerequisite and suggested reading materials.

Section 1 introduces RPG IV by explaining its basic features and illustrating them with a simple example.

Section 2 gives system concepts on hardware and software.

Section 3 details the RPG IV language. It explains the types of statements and their coding, and gives the format and use of each type of statement in the language.

Section 4 explains the use of the hardware in running RPG IV programs.

Section 5 gives an advanced sample program. This program illustrates uses of the instructional material given in previous chapters.



## SECTION 1 — INTRODUCTION

### STRUCTURE OF RPG IV PROGRAMS

RPG IV programs comprise two sequences of statements. First, there is a sequence of **data-defining statements** that defines the structures and formats of the data to be processed. This is followed by a sequence of **procedural statements** that processes the data through the structures defined in the first sequence. This processing yields the output in the desired form. Figure 1-1 shows the general layout of a typical program in RPG IV.

These two sequences of statements handle tables and records, update files, produce reports, and can deal with any other business-oriented applications. Section 3 details the data-defining and procedural statements, and gives their individual formats and uses.

In addition to the program itself, there is the data to be processed. This is a separate sequence of input that has the format(s) specified by the data-defining statements of the program.

Page 1-10 of this section gives a simple RPG IV program and a brief explanation. However, it also aids in understanding the discussions of the statements given in Section 3.

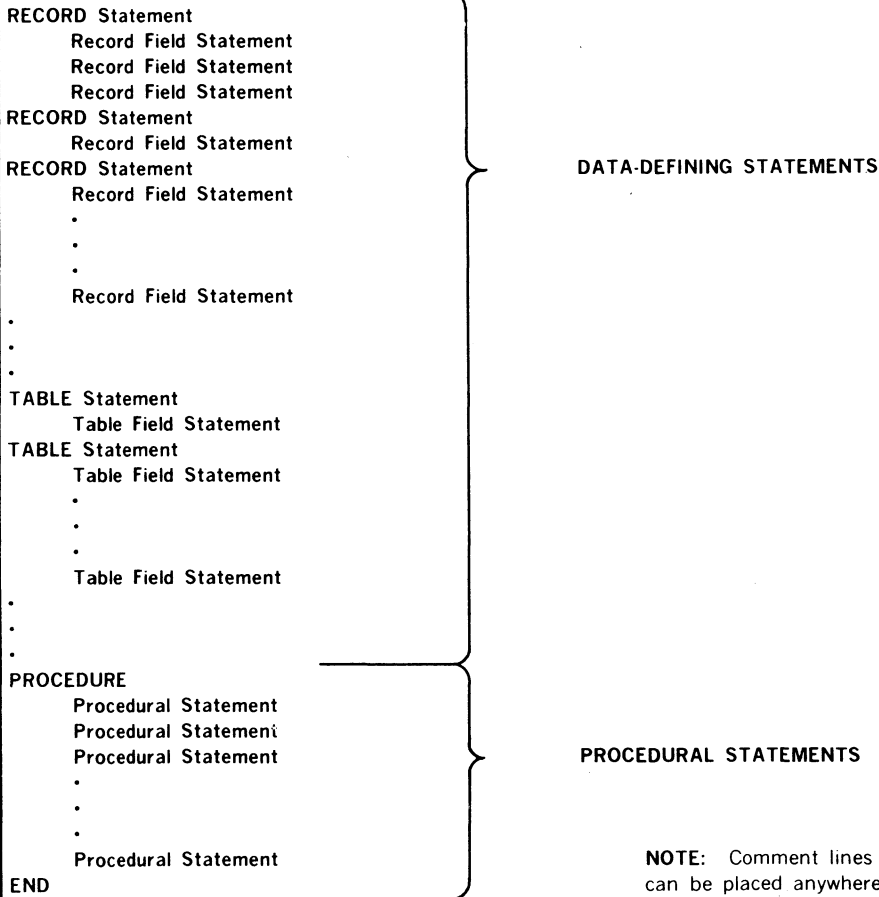


Figure 1-1. Layout of a Typical RPG IV Program

A **statement**, whether it is a data-defining or a procedural statement, is one line of information written in the RPG IV language. It consists of 80 characters corresponding to the 80 columns of a standard punched card.

RPG IV ignores columns 70 through 80. Use these columns for statement identification, comments, or programming aids as desired.

#### NOTE

The data used with the program can be in any column, following the specifications of the data-defining statements.

The statement itself is in columns 1 through 69. It comprises elements defined in the following section arranged in a format that depends on the individual statement (section 3). Regardless of the elements contained in a statement or the format requirements within a statement, each statement is freeform, i.e., there are no requirements for spacing, indentation, or column use within columns 1 through 69. You need no special coding forms.

## DATA-DEFINING STATEMENTS

As indicated in figure 1-1 there are four types of data-defining statements that provide a definition of the data structures to be used by the program. These data structures are records and tables. **Records** hold intermediate results and data being input from or output to files. **Tables** contain related, repetitive data items.

Both records and tables are divided into fields. **Fields** are the elementary variables of any RPG IV program. The computations performed by the program, its logic, and its final output are based on the manipulation of fields and their contents.

A **record statement** identifies a record and specifies the conditions under which this record is manipulated.

A **record field statement** identifies and defines all of the fields in the record. All record field statements pertaining to a given record immediately follow the record statement for that record.



## introduction

A **table statement** identifies a table and specifies its size.

A **table field statement** identifies and defines all of the fields in the entries in a table. All table field statements pertaining to a given table immediately follow the table statement for that table. Each entry in a given table has the same field structure as any other entry in that table.

The formats, elements, and uses of data-defining statements are explained in Section 3.

## PROCEDURAL STATEMENTS

As indicated in figure 1-1 procedural statements follow the data-defining statements. They direct the execution of the program as it processes the data previously defined by the data-defining statements.

The **PROCEDURE** statement is the first procedural statement. It is not executable, but merely serves as the divider between the data-defining statements and the procedural statements. It terminates the processing of data definitions and begins the processing of procedural manipulations. This statement has only one form: the single word **PROCEDURE**.

Subsequent procedural statements manipulate the data to obtain the desired output. Their formats, elements, and uses are explained in Section 3.

Procedural statements are executed in the order of their appearance in the program unless the specified condition is not met, or unless the program is directed to another statement by a **GO TO** or **PERFORM** statement.

The **END** statement is the last procedural statement. It is not executed, but merely serves as a signal that the program is finished. This statement has only one form: the single word **END**. It is the last statement in the program.

## ELEMENTS OF RPG IV STATEMENTS

RPG IV statements contain combinations of the following elements arranged in formats given in section 3:

- a. Statement numbers
- b. Names
- c. Conditions

- d. Indicators
- e. Constants
- f. Literals
- g. Expressions
- h. Comments

The elements are defined in this section and their uses illustrated in the sample program (page 1-10). Section 3, in giving the format of each RPG IV statement, exhaustively explains the application of these elements in the statements.

## STATEMENT NUMBERS

A **statement number** from 1 to 9999 can begin any procedural statement. It identifies the statement so that the program has access to it as required (e.g., in program loops, jumps, and conditional processing). Statements that do not require other than sequential access need not be numbered.

## NAMES

A **name** identifies data or a subroutine referenced by the program. It comprises one to six alphanumeric characters (numbers and letters), the first of which is alphabetic. No blanks are allowed.

### Examples:

```
A  
X15  
FIELD3  
J7U5
```

A **record name** identifies an area of memory that provides space for the characters comprising the record. A record name is assigned by a RECORD statement.

A **table name** identifies an area of memory that provides space for a series of data entries, i.e., a table. All entries in a given table have identical field assignments. A table name is assigned by a TABLE statement.

A **field name** identifies a contiguous set of character positions in a table entry or record. It is assigned by a FIELD statement.

An **implied field name** identifies a field not named in a FIELD statement. The appearance of an implied field name in a procedural statement causes assignment of a memory area to it. This area is large enough to hold any character string or number entered.

## introduction

A **subroutine name** identifies a special procedure outside the program. This procedure (subroutine) performs one of many special functions.

The above types of names conform to the format given at the beginning of this subsection. In addition, there are qualified and subscripted names that take modified formats.

A **qualified name** identifies a field and its record or table so that it is not confused with a like-named field in another record or table. A qualified name consists of a record or table name, a period, and a field name. No blanks are allowed.

### Examples:

```
UPDATE . HOURS  
OUTPUT . HOURS  
C45Y . Y7  
J289RR . Y7
```

A **subscripted name** identifies a table entry. It consists of a table name followed by a computational expression (page 1-8) in parentheses. No blanks are allowed. The integral portion of the result of the computation is the number of the table entry referenced.

### Examples:

```
ACCNT ( INDEX )  
OUTPUT . NAME ( 3 )  
VALUE7 ( X+3 )  
NORTH . Y3 ( X*2 . 53 )
```

## CONDITIONS

A **condition** can be imposed on any procedural statement by placing the condition in parentheses in front of the statement. Such a statement is executed only if the condition is met when the statement comes up for execution.

### Example:

```
( TIME=10 ) MOVE INPUT . JOB , OUTPUT . JOB
```

is a MOVE statement that moves the contents of the field JOB in record INPUT to the same field in record OUTPUT only if TIME has the value 10 when the program is ready to execute this instruction. Time is defined in the program prior to this statement.

If the statement is numbered, the condition follows the number.

**Example:**

```
100 (TIME=10) MOVE INPUT.JOB,OUTPUT.JOB
```

**INDICATORS**

An **indicator** is a program switch that can be on or off. By program switch we mean that the program itself, not part of the computer hardware, turns the switch on or off. You can thus use these switches to control the operations performed by your program, and specify the conditions under which these controls are activated. For instance, you can specify that certain statements be executed if, and only if, certain indicators are on or off.

RPG IV has three types of indicator: general, control break, and special. The indicators and their uses are given in appendix A for later referencing convenience. Refer to this appendix as you study this section.

A **general indicator** is turned on or off by a general procedural statement (Section 3). There are 99 general indicators, specified in coding by the symbols #1 through #99. Used in procedural statements, they specify conditions to be met for the execution of any part of the program. For example

```
(#2) ENTER INPUT, TABLEA
```

enters INPUT in TABLEA only if general indicator #2 is on; and

```
(#1 AND NOT #65) ENTER INPUT, TABLEA
```

enters INPUT in TABLEA only if general indicator #1 is on *and* general indicator #65 is off (the conditions under which these indicators are on or off will have previously been specified by your general procedural statements). All general indicators are off until you write a general procedural statement that turns them on. Of course, you can turn them off again with other procedural statements.

A **control break indicator** is used in the direct updating of a record control field and as a condition for statement execution. There are ten levels of control break: #C1 through #C10.

A **special indicator** is used like a general indicator except that the conditions are implicit in the indicator itself and are not otherwise specified. A special indicator is turned on or off by a general procedural statement, or by one of the explicit procedural statements SET ON, SET OFF, or SET conditional (Section 3). For example:

```
(#") ENTER INPUT, TABLEA
```

## introduction

enters INPUT in TABLEA only if the previous statement was executed. One frequent use of # " is to repeat long or complicated conditions for successive statements.

### NOTE

All special indicators except #OV are off until a statement that turns them on is executed. However, because #OV is normally used to begin a new report page, it is initially on.

## CONSTANTS

A **constant** is a positive number used by the program. It can contain up to 14 significant decimal digits before, and nine after, the decimal point. For integer constants, no decimal point is required. No blanks are allowed.

### Examples:

```
375.125
100
.0333333
12345678901234.123455789
```

## LITERALS

A **literal** is a string of alphanumeric characters (on one line) used by the program. A literal is enclosed within apostrophes. Blanks are allowed in literals.

### NOTE

If an apostrophe is to be part of a literal, use two consecutive apostrophes.

### Examples:

```
'THIS IS A LITERAL'
'DON'T'
'$350.00'
```

## EXPRESSIONS

An **expression** in RPG IV is one of three types: computational, relational, or conditional.

A **computational expression** is a combination of constants and/or numeric fields with the arithmetic operators + (addition), - (subtraction), \* (multiplication), and / (division). In a computational expression, operations within parentheses are performed first, and multiplication and division are performed before addition and subtraction. Within these levels, operations are performed from left to right. After nine digits to the right of the decimal place in the result of the computation, there is rounding for multiplication and division, and truncation for addition and subtraction.

**Examples:**

$$A+B*C$$

multiplies B by C and adds A;

$$(X*Y) + (U*V)$$

multiplies X by Y and U by V, and adds the results;

$$-VAL1(J)/37.5$$

divides VAL1(J) by 37.5 and negates the result; and

$$-(A*(B+C+D+E))$$

multiplies A by the sum of B, C, D, and E, and negates the result. Numeric fields in a computational expression can contain editing characters and embedded blanks without affecting the arithmetic interpretation since only the digits, sign, and decimal point are significant. Thus, a field with two implied decimal places could contain either 00002575 or \$\*\*25.75 and be interpreted as 25.75 in the computation.

A **relational expression** compares two computational expressions, or two alphanumeric fields or literals, for a specific relational condition. The two expressions, fields, or literals are separated by one or two of the relational operators <, =, and >, as follows:

|           |                                          |
|-----------|------------------------------------------|
| <         | Less than                                |
| >         | Greater than                             |
| =         | Equal to                                 |
| < = or =< | Not greater than (less than or equal to) |
| > = or => | Not less than (greater than or equal to) |
| <> or ><  | Not equal to (less than or greater than) |

If the relation is true, the condition is met. If not, the condition is not met.

**Examples:**

$$FIELD A \leq FIELD B$$

states that the condition is met when FIELD A is less than or equal to FIELD B;

$$A*B > 10$$

states that the condition is met when A\*B is greater than 10; and

$$A1(INDEX) >< LIMIT$$

## introduction

states that the condition is met when A1(INDEX) is not equal to LIMIT. Note that a constant or implied numeric field cannot be compared with an alphanumeric field or literal.

A **conditional expression** combines indicators and/or relational expressions with the logical operators AND, OR, and NOT to form a logical condition. In conditional expressions, operations within parentheses are performed first, and NOTing is performed before ANDing, which is performed before ORing. Within these levels, operations are performed from left to right. NOT can follow another logical operator. At least one blank follows each logical operator if the next character of the statement is a letter or digit.

### Examples:

# 1 AND NOT # 2

states that the condition is met only if indicator # 1 is on *and* indicator # 2 is off;

NOT A>B OR #OV

states that the condition is met only if A is not greater than B or if the page overflow indicator #OV is on.

## COMMENT LINES

A **comment line** improves the format of the listing or documents the program. It appears in the listing but neither acts nor is acted upon. A comment line is either entirely blank or has an asterisk as the first nonblank character. Blanks are allowed.

### Examples:

\*REMARKS CAN OFTEN CLARIFY A PROGRAM  
\*THIS IS A COMMENT \$85+N

## SAMPLE PROGRAM

This section shows a flowchart (figure I-2), an RPG IV program (figure I-3), the data to be processed by the sample program (figure I-4), and the resulting report (figure I-5). It would serve little purpose here to give a detailed description of the total operational sequence involved in this data processing, since this sample is intended to serve as a reference guide while reading the material in Sections 2 and 3, and to show how the program, data, and results are related.

Notice that the first data-defining statements are a group of literals under the record named HEAD. The first procedural statement reads a card. If it is the first card (#F on),

the output control fields are initialized by the MOVE statement. If this card would cause page overflow on printing, the printer goes to the top of the next page, increments the page NO, and prints the heading HEAD (the overflow indicator #OV is also on at the beginning of a program since it is assumed that the first line of output will be at the top of the page). Thus, the first printer action, PRINT if #OV is on, skips to the top of the next page (\$C1), prints HEAD, and skips a line (\$A1). Note that the record named HEAD is printed as is, with DEPARTMENT in character positions 16 through 25. ACCOUNT number in 29 through 42, employee in 47 through 54, hours in 59 through 63, PAGE in 72 through 75, and the page number (i.e., the value of NO) in 77 and 78 as specified in the data-defining statements under the RECORD HEAD card.

Now that the heading is printed, the program considers the card already read. Since the READ CARD procedural statement specifies the record named INPUT, the data on the card are placed into the record INPUT according to the format given by the data-defining statements of that record. Thus, the record field DEPT comprises character positions 1 through 3 of the record, EMPNO 4 through 9, ACCT 10 through 24, and HOURS 75 through 80. The designation 80.1 specifies that the record field HOURS has one place to the right of an implied decimal point. Examination of the first data card shows that the data are in compliance with the specifications of the record named INPUT. Thus, 10A in card columns 1 through 3 is the department number DEPT, 210356 in card columns 4 through 9 is the employee number EMPNO, etc. Upon printing, the positions of these fields shift to those in the record named OUTPUT so that the department number is printed in character positions 20 through 22, etc. The symbols #C1, #C2, and #C3 are control break indicators that in this case specify that an employee number be printed for every entry, but that account and department numbers be printed only for the first applicable entry.

Further details of the procedure will become apparent on further study and the reading of Sections 2 and 3.



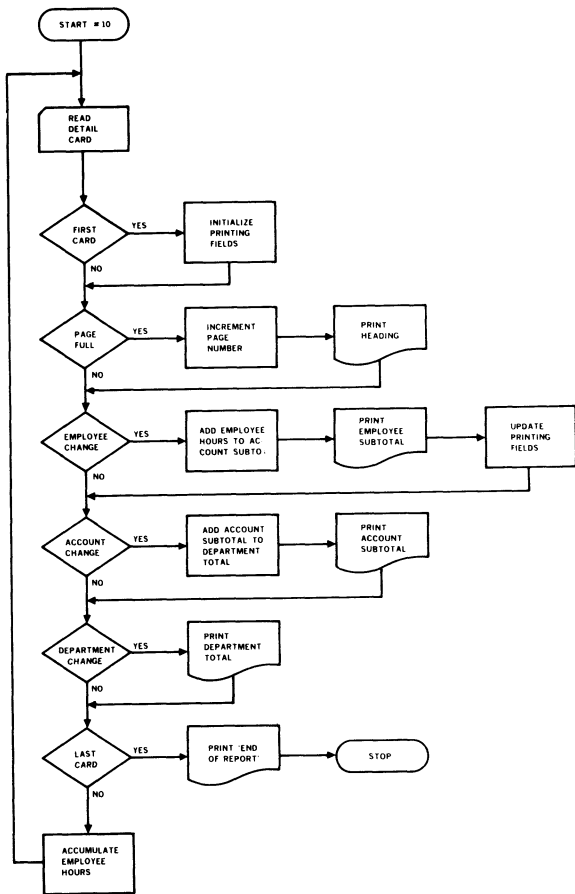


Figure 1-2. Flowchart for the Sample RPG IV Program

## VARIAN RPG IV SOURCE LISTING

```

*
*      SAMPLE RPG IV PROGRAM
*

RECORD HEAD
    (16,25)  'DEPARTMENT'
    (29,42)  'ACCOUNT NUMBER'
    (47,54)  'EMPLOYEE'
    (59,63)  'HOURS'
    (72,75)  'PAGE'
    NO(77, 78.0) Z
RECORD INPUT
    DEPT(1,3) #C3
    EMPNO(4,9) #C1
    ACCT(10,24) #C2
    HOURS(75,80.1)
RECORD OUTPUT
    DEPT(20,22) B,P#C3
    ACCT(29,41) B,P#C2
    EMPNO(48,53) P#C1
    EHOURL(58,63.1) B,Z,D
RECORD SUBTOT
    AHOURS(66,71.1) B,Z,D
RECORD TOTAL
    DHOURL(73,78.1) B,Z,D
PROCEDURE
10  READ CARD INPUT
    (#F)  MOVE INPUT.DEPT,OUTPUT.DEPT
    (#")  "    INPUT.ACCT,OUTPUT.ACCT
    (#")  "    INPUT.EMPNO,OUTPUT.EMPNO
    (#OV) COMPUTE NO=NO+1
    (#")  PRINT $C1,HEAD,$A1
    (#C1) COMPUTE AHOURS=AHOURS+EHOURL
    (#")  PRINT OUTPUT
    (#")  POST INPUT,OUTPUT
    (#C2) COMPUTE DHOURL=DHOURL+AHOURS
    (#")  PRINT SUBTOT
    (#C3) PRINT TOTAL
    (#LC) STOP 'END OF REPORT'
        COMPUTE EHOURL=EHOURL+HOURS
        GO TO 10
        END

```

Figure I-3. Sample RPG IV Program

|                      |      |
|----------------------|------|
| 10A210356ALPHA-29107 | 1200 |
| 10A210356ALPHA-29107 | 800  |
| 10A350017ALPHA-29107 | 392  |
| 10A350017ALPHA-29107 | 1500 |
| 10A151179BETA-35     | 800  |
| 10A161711BETA-35     | 800  |
| 10A290238BETA-35     | 800  |
| 10A750192BETA-35     | 800  |
| 25B019372HOUSE-1997  | 505  |
| 25B019372HOUSE-1997  | 405  |
| 25B019372HOUSE-1997  | 50   |
| 25B317911HOUSE-1997  | 1500 |
| 25B607712HOUSE-1997  | 1200 |

Figure I-4. Data for the Sample RPG IV Program

| DEPARTMENT    | ACCOUNT NUMBER | EMPLOYEE | HOURS | PACE  | 1 |
|---------------|----------------|----------|-------|-------|---|
| 10A           | ALPHA-29107    | 210356   | 200.0 |       |   |
|               |                | 350017   | 189.2 |       |   |
|               |                |          |       | 389.2 |   |
|               | BETA-35        | 151179   | 80.0  |       |   |
|               |                | 161711   | 80.0  |       |   |
|               |                | 290238   | 80.0  |       |   |
|               |                | 750192   | 80.0  |       |   |
|               |                |          |       | 320.0 |   |
| 25B           | HOUSE-1997     |          |       | 709.2 |   |
|               |                | 019372   | 96.0  |       |   |
|               |                | 317911   | 150.0 |       |   |
|               |                | 607712   | 120.0 |       |   |
|               |                |          |       | 366.0 |   |
| END OF REPORT |                |          |       | 366.0 |   |

Figure I-5. Final Report from Sample RPG IV Program

## SECTION 2 – SYSTEM CONCEPTS

### HARDWARE

The hardware components of the *Varian 620/RPG IV System* include a Varian 620-series computer with at least 4,096 (4K) words of memory, a 620-25 card reader for entering data, a 620-77 line printer for producing finished reports, and a 620-27 card punch to record intermediate information and to aid in the preparation of the programs that process the data.

For information on the hardware system, refer to the Varian documentation on the 620/i, 620/L, or 620/f computer and the individual peripherals.

Section 4 explains to the computer operator how RPG IV programs are run on the system.

### SOFTWARE

You provide two pieces of software to produce reports on the *Varian 620/RPG IV System*:

- a. You write a **program** according to the directions in Section 3. This program comprises two parts: data-defining statements to specify the forms that the data to be processed will take, and procedure statements to specify how the data in the defined forms will be processed.
- b. You supply **data** according to the specifications given in the data-defining portion of your RPG IV program.

The software supplied with the *Varian 620/RPG IV System* processes the data you have supplied according to the specifications of the program you have written.

The basic software component of the system is the **RPG IV two-part compiler**. This component compiles your program and yields an **object deck**. The object deck, the **RPG loader**, and the **RPG runtime** support program process your data. The use of these software components is explained in section 4. All except the object deck are supplied with the *Varian 620/RPG IV System*. (The object deck, of course, is the output of the compilation of your program by the supplied RPG IV two-part compiler.)



## SECTION 3 – RPG IV STATEMENTS

### DATA-DEFINING STATEMENTS

Data-defining statements are the first statements in an RPG IV program. They provide a definition of the data structures (records and tables) to be used by the program and processed according to the procedural statements (page 3-10).

In the descriptions of the statement formats, **boldface** type designates required items and *italic* type designates optional items. Items in capital letters are coded just as written. Items in lower-case letters represent variables, constants, values, etc.

The data-defining statements are divided into *record statements*, *record field statements*, *table statements*, and *table field statements*.

A **record statement** identifies a record and specifies the condition under which this record is manipulated. It is followed by the **record field statements** pertaining to the fields of this record, identifying and defining the fields. A **table statement** identifies a table and specifies its size. It is followed by the **table field statements** pertaining to the fields of the entries in this table, identifying and defining them.

#### RECORD STATEMENT

A **record statement** identifies a program record and its area in memory. It contains identifiers that specify the selection criteria (pages 3-2 and 3-3) for data to be input to the record by a READ CARD procedural statement (page 3-20) and an indicator that is turned on when data are input to the record and turned off when it is not. The format of a record statement is

**RECORD name** (*identifier,identifier,...*)*indicator*

where **name** identifies the record and its area in memory, *identifier* is a record selection code, and *indicator* is a symbol for an indicator that is turned on when data are accepted and input into the record or off when the data are rejected.

For example, the record statement

**RECORD GAIN (80C1)#17**

identifies the record named GAIN, specifies that a READ CARD statement can enter data in this record only when column 80 of the data card contains a one, and turns indicator # 17 on if data are accepted or off if data are rejected.

If the record statement contains no selection criterion (*identifier*), any READ CARD statement that references this record will input data to the record.

## Identifier

The **identifier** in a record statement is a record selection code that specifies the criteria for the input of data to this record by a READ CARD statement (page 3-20). If the criteria specified by the identifier are met, the data from the card are input to the specified record and the indicator designated in the RECORD statement is turned on. Identifiers are enclosed in parentheses.

An identifier has one of the following formats:

**pCx pDx pZx pNCx pNDx pNZx**

where C, D, and Z specify that the selection is based on the entire character (C), the digit (0-9 punches) portion (D), or the zone (11-12 punches) portion (Z); p is a number from one to 80 specifying the card column used for comparison; x is a character punched in that column; and N (NOT) specifies that the comparison must fail for data to be accepted into the record.

For example, the identifier 1C3 specifies that the character in column 1 of the data card must be a three for data to be accepted into the record. Identifier 80NZA specifies that for data to be accepted into the record, the character in column 80 of the data card *cannot* have the zone bits of the character A (i.e., since A has a 12-punch as a zone bit, the character in column 80 must have a different zone bit configuration than a 12-punch).

## Multiple Identifiers

Record statements can contain multiple identifiers to specify ANDed or ORed conditions for the acceptance of data into the record.

A sequence of identifiers separated by commas and included within one set of parentheses ANDs the selection criteria of the individual identifiers. For example, the sequence

**( 15CX, 20NZ- ) #79**

specifies that for acceptance of data into the record, there is an X in column 15 *and* there cannot be the zone bits of the minus sign in column 20. The indicator is given at the conclusion of the identifier sequence.

A sequence of identifiers in separate subsequences, separated by commas and indicator symbols, ORs the selection criteria of the individual identifiers. For example, the sequence

**( 80C1 ) #17, ( 80C2 ) #27**

specifies that for acceptance of data into the record, column 80 contains a one or a two. The sequence

**( 22D\$ ) #40 , ( 54CE ) #40**

Specifies that for acceptance of data into the record, column 22 contains the numeric bits of the \$ (i.e., a 3-8 punch) or column 54 contains an E. An indicator is given after each identifier.

These specifications can be combined. For example, the sequence

**( 1Z- ) #1 , ( 1NZ- , 2C ) #2**

specifies that for acceptance of data into the record, column 1 contains the zone bits of the minus sign, or if column 1 does not contain the zone bits of the minus sign and column 2 is blank.

## RECORD FIELD STATEMENTS

All **record field statements** for a given record directly follow the record statement. They define all fields in the record. The fields can appear in any order and can overlap.

The *record field statement for an alphanumeric field* has the format

*field (first,last),parameter,parameter...*

where *field* is the name (if any) of the field, **first** is the number of the first character position in the field, **last** is the number of the last character position in the field, and *parameter* is one of the parameters discussed on page 3-4.

Statement-identifying names and logical operators (e.g., RECORD, MOVE, NOT) cannot be used as record field names.

If more than one parameter is required, enter additional parameters, separated by commas, following the first *parameter*. Parameters can appear in any order.

The *record field statement for a numeric field* has the format

*field (first,last.decimal),parameter,parameter,...*

where the definitions are as above except that **last** is followed by a decimal point and **decimal**, which specifies the number of digits to the right of the implied decimal point in the field. It is present for every numeric field, even if the value is zero. If the field contains an actual decimal point, its position overrides the specification in the record field statement.



## statements

An example of a record field statement for an alphanumeric field is

**( 16 , 25 ) , ' DEPARTMENT '**

which places the literal DEPARTMENT in character positions 16 through 25 of the record to which the record field statement applies.

An example of a record field statement for a numeric field is

**E HOURS ( 59 , 64 . 1 )**

which places data having one place to the right of the decimal point in the field E HOURS. This field occupies character positions 59 through 64 of the record to which the record field statement applies.

## Parameters

The following parameters can be used with record field statements to define more closely the format of the data. Parameters can appear in any order. Each parameter is preceded by a comma.

### BLANK (B PARAMETER)

The **B parameter** consists of the letter B. It indicates that the field is to be cleared (blanked) after the record is output with a PRINT (page 3-21) or PUNCH (page 3-22) statement. An example of a record field statement containing this parameter is

**ACCNT ( 10 , 19 ) , B**

### CONDITIONAL POSTING (P PARAMETER)

The **P parameter** consists of the letter P followed by an indicator that is on for conditional posting. If the indicator is off when a POST (page 3-13) or COLLECT (page 3-13) operation would normally modify this field, no modification occurs. An example of a record field statement containing this parameter is

**ACCNT ( 10 , 19 ) , P # 1**

## EDITING PARAMETERS

An **editing parameter** consists of one of the letters or symbols listed below. It edits a numeric field according to the corresponding explanation. More than one editing parameter can be used in a record field statement, but each must be separated by commas.

- H Half-round the low-order digit.
- Z Suppress leading zeros.
- D Insert actual decimal point.
- C Insert commas.
- \$ Insert \$ before first digit.
- \* Replace leading zeros with asterisks.
- Allow one position to the right for a minus sign.
- CR Allow two positions to the right for a credit sign.

Editing parameters do not apply to alphanumeric fields. Examples of record field statements containing editing parameters are

```
EHOURS ( 59,64.1 ),D
EHOURS ( 59,64.1 ),B,Z,D
BAL ( 35,44.2 ),$,*,C,D,CR
```

## AUDITING PARAMETER

The **auditing parameter** consists of any combination of the letters A, N, and S; followed (optional) by any combination of the letters R, L, and J; followed (obligatory) by an indicator symbol. The auditing parameter checks the validity and positioning of characters in a field.

The first set of letters indicates the characters permitted by the audit:

- A permits alphabetic characters
- N permits numerals
- S permits special characters

These can be combined. For instance, AN permits alphabetic characters and digits, but no special characters. The presence of a nonpermitted character causes the audit to fail.

## statements

The second set of letters, if used, indicates the allowed positions of the permitted characters:

- R right-justified (rightmost character nonblank)
- L left-justified (leftmost character nonblank)
- J all characters juxtaposed (no embedded blanks)

If none of these letters appears in the auditing parameter, all positions are permitted for characters and blanks.

The indicator following the letters is turned on if the audit passes and off if the audit fails. For example, the auditing parameter NRJ #20 turns indicator #20 on only if the rightmost characters are digits and there are no nonnumeric characters or embedded blanks (leading blanks are allowed).

**Special case:** The letter J alone with an indicator symbol causes an audit for an all-blank field.

Examples of record field statements containing auditing parameters are:

```
NAME ( 21, 30 ), AL, #40
ZAP ( 1, 69 ), J#1
NUMBER ( 1, 10 ), N#99
CLOSED ( 11, 27 ), ANSLJ#3
```

## CONTROL PARAMETER

The **control parameter** consists of one of the control break indicators #C1 through #C10. A record field statement containing a control parameter defines a **control field**.

Whenever there is a direct updating (page 3-7) of a control field, there is a check for a control break. A **control break** occurs when there is a direct updating of a control field that changes the contents of that field. When such a change takes place, the corresponding control break indicator and all lower-numbered control break indicators are turned on.

However, if the contents of the control field are unchanged by the direct updating, the corresponding control break indicator only is turned off. The lower-numbered control break indicators remain unchanged from their previous states.

The first time that there is a direct updating of a control field, the indicator #F, rather than a control break indicator, is turned on. Any subsequent updating turns indicator #F off and the corresponding control break indicator(s) on.

When a READ CARD statement (page 3-20) that references a record with a control field encounters the last data card (which has /\* in columns 1 and 2), a control break for that field occurs.

### NOTE

Data placed in one of a set of overlapping fields does not constitute a direct updating of the other fields. Such other fields are thus not checked for a control break even though their contents can be changed by the new data.

### SEQUENCE PARAMETER

The **sequence parameter** consists of one of the characters >, =, or < followed by an indicator symbol. It specifies that the field is to be checked for sequence during a direct updating (see below). The specified indicator is turned on if the new contents of the field are greater than, equal to, or less than, respectively, its previous contents. All fields are initialized to blank (lowest value of the collating sequence) unless explicitly set as a literal field.

For example, the sequence parameter > #50 specifies that indicator #50 is turned on when a direct updating of the field increases the value of its contents. If the direct updating does not change the contents of the field, or decreases them, indicator #50 is turned off.

### NOTE

Data placed in one of a set of overlapping fields does not constitute a direct updating of the other fields. Such other fields are thus not checked for sequence even though their contents can be changed by the new data.

### LITERAL PARAMETER

The **literal parameter** consists of a literal used to initialize a field. Positions not initialized by the literal are cleared (i.e., contain blanks).

#### Examples:

```
( 5 , 17 ) , 'TOTAL HOURS='
( 30 , 40 ) , 'TOTAL COST='
```

### Direct Updating of Record Fields

A **direct updating of a record field** occurs whenever:

The *table field statement for an alphanumeric field* has the format

**field (first,last),post,KEY**

where **field** is the name of the field, **first** is the number of the first character position in the field, **last** is the number of the last character position in the field, **post** is the letter P plus a general indicator used for conditional posting like the P parameter of a record field statement (page 3-4), and **KEY** specifies that the table is sequential and that this is the key field in the table.

**KEY** in a table field statement indicates the **key field** of a sequential table. The table field statement containing **KEY** is the field used for searching by procedural statements. **KEY** can be used for only one field per table. If **KEY** is not present in any table field statement for a given table, it is a LIFO (last-in-first-out) table.

The *table field statement for a numeric field* has the format

**field (first,last.decimal),post,KEY**

where the definitions are as above except that **last** is followed by a decimal point and **decimal**, which specifies the number of digits to the right of the implied decimal point in the field. It is present for every numeric field, even if the value is zero. If the field contains an actual decimal point, its position overrides the specification in the table field statement.

An example of a table field statement for an alphanumeric field is

**ACCNT ( 1 , 10 ) , KEY**

which specifies that the field **ACCNT** occupies character positions one through ten in each entry in the table, and that **ACCNT** is the key field for this table.

An example of a table field statement for a numeric field is

**AMOUNT ( 11 , 17 . 2 ) , P # 32**

which specifies that the field **AMOUNT** occupies character positions 11 through 17 in each entry in the table, and that there are two digits to the right of the implied decimal point in the field. Indicator **#32** is the conditional posting indicator. This field is not a key field.

## PROCEDURAL STATEMENTS

Procedural statements follow the data-defining statements. They direct the execution of the program as it processes the data previously defined by the data-defining statements.

The **PROCEDURE** statement is the first procedural statement and comprises the single word PROCEDURE. It serves as the divider between the data-defining statements and the procedural statements.

Subsequent procedural statements manipulate the data to obtain the desired output. They have the general form

*statement number (condition) VERB direction*

where the optional *condition* specifies the condition(s) under which the statement is to be executed, **VERB** specifies the action to be taken, and **direction** specifies the object(s) of the verb. If no condition is specified, the statement is executed unconditionally. The formats, elements, and uses of individual procedural statements are explained in the following subsections. Any procedural statement can begin with a statement number (optional).

The **END** statement is the last procedural statement. It is invariable and terminates the program.

If the verb of a procedural statement is to be repeated in subsequent statements, it can be replaced by the ditto mark (").

**Example:**

```
( #3 OR #44 ) MOVE C,D
( #3 or #44 ) " A, B
```

The ditto mark cannot be used for repetition of directions.

The # " indicator can be used for repetition of a condition.

**Example:**

```
( #3 OR #44 ) MOVE C,D
( # " ) " A, B
( # " ) COMPUTE X = J+TOTAL
```

## PROCEDURE STATEMENT

This statement is always the first procedural statement. Thus it directly follows the last data-defining statement and serves as a divider between the two types of statement. PROCEDURE terminates the processing of data definitions and begins the processing of procedural manipulations. This statement has only one form:

## PROCEDURE

## MOVE STATEMENT

This statement moves a literal, a constant, or the contents of one field to another field or fields. It has the format

(condition) **MOVE from,to,to,...**

where **from** is the literal or constant to be moved, or the name of a field whose contents are to be moved; and **to** is the name of the field to receive the moved item. The movement can be made to additional fields by entering the names of all such fields, separated by commas, in the **MOVE** statement after the first **to**.

A **numeric movement** occurs when the from-field contains a constant or the name of a numeric field, and the to-field is a numeric or implied field. A numeric movement moves only numeric information, re-editing and rescaling it to the format of the to-field.

An **alphanumeric movement** occurs in all other cases. However, constants or implied numeric fields cannot be moved to explicit alphanumeric fields. An alphanumeric movement moves the characters one by one, left to right. Movement stops when the to-field is full. If the from-field is shorter than the to-field, the to-field is filled out with blanks after all characters have been moved from the from-field.

## Examples:

```

                                MOVE 'TITLE',FIELD1,FIELD2,FIELD3
(TIME=10)                      MOVE INPUT.JOB,OUTPUT.JOB
                                MOVE INVAL,OUTVAL

```

## MOVEZ STATEMENT

This statement moves *only the zone bits* of the characters in a literal or field to the zone bits of corresponding characters in another field or fields. It has the format

(condition) **MOVEZ from,to,to,...**

where **from** is the literal or field whose zone bits are to be moved, and **to** is the name of the field to receive the moved zone bits on corresponding characters. The movement can be made to additional fields by entering the names of all such fields, separated by commas, in the **MOVEZ** statement after the first **to**.

The **MOVEZ** statement does not apply to numeric movements. It operates as an alphanumeric movement under a **MOVE** statement except that only the zone bits of the characters are moved. (Zone bits are those corresponding to the 11- and 12-punches on punched card input.)

**Examples:**

```
( # 1 OR X=5 )      MOVEZ 'A',XYZ
                     MOVEZ NEWSUM, CODE1, CODE4
```

**POST STATEMENT**

This statement posts the contents of fields in one record or table entry to *like-named* fields in other records or entries. It has the format

(condition) **POST from,to,to...**

where **from** is the name of the record or entry from which the posting is made, and **to** is the name of a record or entry to which the posting is made. The posting can be made to additional records or entries by entering the names of all such records or entries, separated by commas, in the POST statement after the first **to**. Posting obliterates the original contents of the to-fields, and replaces them with the contents of the from-field.

The field moves as under a MOVE statement except when the field in the to-record or to-entry has a P parameter that is off (page 3-4).

**Examples:**

```
( #C3 )      POST INPUT, OUTPUT
( # " )      POST INPUT, OUTPUT, TOTAL
              POST BOOKS ( 2 * X ), CURRENT
```

**COLLECT STATEMENT**

This statement adds the contents of fields in one record or table entry to the contents of *like-named* fields in other records or entries. It has the format

(condition) **COLLECT from,to,to...**

where **from** is the name of the record or entry whose contents are to be added, and **to** is the name of a record or entry whose contents are to be augmented by the amount contained in **from**. The addition can be made to additional records or entries by entering the names of all such records or entries, separated by commas, in the COLLECT statement after the first **to**. The COLLECT statement functions like the POST statement except that the to-fields after execution contain the sum of their former contents and the contents of the from-field.

The field moves as under a MOVE statement except when the field in the to-record or to-entry has a P parameter that is off (page 3-4). If any significant digits of the result are lost because the to-field is not large enough to hold them, computational overflow indicator #X1 comes on. If either field is a nonnumeric implied field, mode error indicator #X2 comes on.



## statements

### Examples:

```
(#17)    COLLECT DETAIL, MASTER  
         COLLECT ACCT, DEPT, TOTAL
```

## ADD STATEMENT

This statement adds a constant or the contents of one field to the contents of other fields. It has the format

(condition) **ADD from,to,to,...**

where **from** is the constant or the name of the field whose contents are to be added, and **to** is the name of the field containing the value to which the addition is made. The addition of **from** can be made to the contents of several fields by entering the names of all such fields, separated by commas, in the ADD statement after the first **to**.

The accuracy of the result depends on the size of the to-field and the position of the decimal point in it. If any significant digits of the result are lost because the to-field is not large enough to hold them, computational overflow indicator #X1 comes on. If either field is a nonnumeric implied field, mode error indicator #X2 comes on.

The ADD statement can be coded as a COMPUTE statement for updating a field as follows:

**COMPUTE to = to+from**

### Examples:

```
(#C1)    ADD MONTH, YTD  
         ADD ACCT.AMT, DEPT.AMT  
         COMPUTE ZAP =ZAP+1
```

## COMPUTE STATEMENT

This statement computes the value of an expression and places the result in the specified field. It has the format

(condition) **COMPUTE field = expression**

where **field** is the name of the numeric or implied field (Section 2) receiving the result of the computation, and **expression** is the computational expression, constant, or numeric field being evaluated.

The result is moved as for a numeric move under a MOVE statement. Editing occurs if it has been specified for the specified field.

When the result is plus, zero, or minus, the corresponding indicator ( #P, #Z, or #M) is turned on and the other two turned off.

Computational overflow turns the #X1 indicator on.

#### Examples:

```
(#C3)      COMPUTE AHOURS = AHOURS+EHOURS
(#2 OR #5) COMPUTE A(I) = 10*B+(2*C)
           COMPUTE DETAIL.COST = RATE*HOURS
           COMPUTE TOTAL = X(1)+X(2)+X(3)+X(4)
```

## GO TO STATEMENT

This statement alters the flow of the program by specifying the next statement to be executed. It has the format

*(condition)* **GO TO number**

where **number** is the number of the next statement to be executed.

#### Examples:

```
(NOT #E)    GO TO 1000
(A=B)       " 777
            GO TO 255
```

## INDEXED GO TO STATEMENT

This statement, like the GO TO statement, alters the flow of the program by specifying the next statement to be executed. It has the format

*(condition)* **GO TO (number,number,...)index**

where **number** is the number of a statement that can be the statement selected by the value of the **index**, which is the name of a numeric field. More than one statement can be included by entering additional statement numbers, separated by commas, after the first statement, but within the parentheses.

The integer portion of the contents of **index** selects the number of the statement to be executed next. Thus, if this value is one, two, three, etc., the first, second, third, etc., statement number, respectively, is the number of the statement to be executed next. For instance, in the first example below, the next statement executed is statement number 20

## statements

if the value of the integer portion of the contents of X is two. If the index is out of range, the program continues in sequence and next executes the statement following the indexed GO TO statement.

### Examples:

|                  |       |                            |
|------------------|-------|----------------------------|
| ( #3 OR NOT #7 ) | GO TO | ( 10,20,30 )X              |
| ( #")            | GO TO | ( 101,35,972,15 )INPUT.KEY |
|                  | "     | ( 11,22,33,500 )UU78       |

## PERFORM STATEMENT

This statement, like the GO TO statement, alters the flow of the program by specifying the next statement to be executed. In addition, however, PERFORM stores the present address of the program so that when a RETURN statement (see below) is found in the sequence following the specified statement, the program flow returns to the main sequence at the statement following the PERFORM statement. The format of the PERFORM statement is

(condition) **PERFORM number**

where **number** is the number of the next statement to be executed. Note that this statement is always used in conjunction with a RETURN statement. If the RETURN statement is missing, the effect of the PERFORM statement is that of a GO TO statement.

### Examples:

|           |                     |
|-----------|---------------------|
| ( A > B ) | <b>PERFORM 95</b>   |
|           | <b>PERFORM 7250</b> |

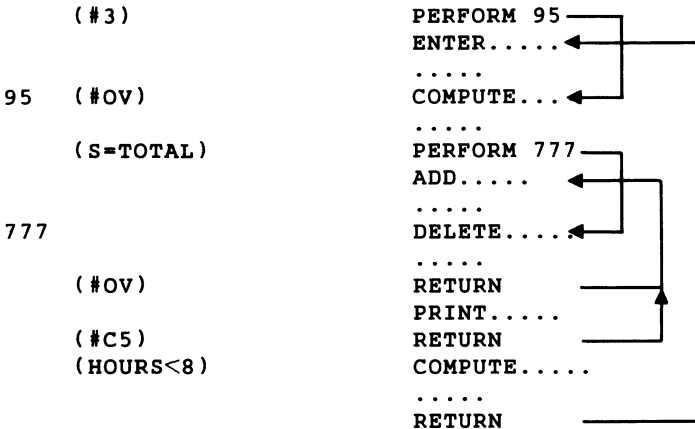
## RETURN STATEMENT

This statement returns the flow of the program to the statement following the corresponding PERFORM statement (see above). The RETURN statement has the format

(condition) **RETURN**

There is at least one RETURN statement for each PERFORM statement, but only one of these RETURN statements is executed on each pass. The PERFORM statement places the return location value in a LIFO (last-in-first-out) queue. The following RETURN statement uses the last such location placed in the queue. Thus, subroutines can be nested using these two statements.

**Example** (arrows show the flow of the program when the condition of the statement is met):



## SET STATEMENT

This statement turns indicators on or off under specified conditions. It has the format

*(condition)* **SET** **value** **indicator**,*indicator*,...

where **value** is the word ON, the word OFF, or a conditional expression within parentheses; and **indicator** is the symbol for an indicator. More than one indicator can be specified by entering their symbols, separated by commas, after the first **indicator**.

If **value** is ON and the statement *condition* is met, the specified indicators are turned on. If **value** is OFF and the statement *condition* is met, the specified indicators are turned off. If **value** is a conditional expression that is true and the statement *condition* is met, the specified indicators are turned ON, but if the condition expression is not true, the specified indicators are turned OFF. In any case, if the statement *condition* is not met, the indicators are unchanged because this statement will not be executed.

If a control break indicator is turned on or off by a SET statement, there is no change in the status of any other control break indicators, i.e., the lower-level control break indicators are unaffected by a SET statement unless they are explicitly specified therein.

### Examples:

```

    (#C1)    SET ON #1, #2, #58
             SET (HOURS<8.0 AND TIME>500) #26
    (#C6)    SET OFF #LC
  
```

## ENTER STATEMENT

This statement assigns space for a new entry to a table and posts data into the new entry. It has the format

(condition) **ENTER** record,table,index

where **record** is the name of the table entry or record from which the posting is made, **table** is the name of the table into which the new entry is being posted, and **index** is the name of a numeric or implied field that is set equal to the number (subscript) of the new entry.

The address of the new entry depends on the type of table (page 3-8):

- a. In a *LIFO (last-in-first-out)* table, the address is one greater than that of the current highest entry or subscript reference value.
- b. In a *sequential table*, the **record** contains a field having the same name as that of the table's key field. The value of this field determines where the new entry is assigned. When this value matches an existing key in the table, the new entry overlays that position. Otherwise, all entries having higher key values move up one position to make room for the new entry.

After the address of the new entry is established, the ENTER statement posts the contents of **record** to that entry just as under a POST statement (page 3-13). Data can also be entered in a table by subscript referencing without the use of ENTER statements.

If the new entry would cause the table to overflow, no posting occurs and the table overflow indicator (a general indicator you will have defined) is turned on. This indicator can be turned off only by a SET OFF statement (page 3-17).

After a successful entry, the implied subscript for the table references the new entry.

**Examples:**

```

ENTER INPUT, TABLEA
( #22 ) ENTER DATA, CTROL, IX55

```

## DELETE STATEMENT

This statement deletes an entry from a table. It has the format

(condition) **DELETE** table(subscript),key

where **table** is the name of the table from which the deletion is to be made; **subscript** (enclosed in parentheses) is the number of the entry to be deleted; and **key** is a field name, constant, or literal used to find the entry to be deleted. A key is used only for sequential tables in the absence of a **subscript**.

When there is neither *subscript* nor *key*, the entry to be deleted is specified by the implied subscript and depends on the type of table (page 3-8):

- a. In a *LIFO (last-in-first-out) table*, the entry deleted is the current highest entry. Deletion reduces the current highest entry by one.
- b. In a *sequential table*, the entry deleted is the last entry entered (page 3-18) or looked up (see below).

In any case, all entries above the deleted entry move down one position in the table after the entry is deleted.

#### Examples:

```

                DELETE TABLEA
( #42 AND #43 ) DELETE CTROL, MONTH
  
```

## LOOKUP STATEMENT

This statement determines, in a sequential table, if there is an entry having a key field (page 3-9) equal to or greater than the key specified in the LOOKUP statement, and sets indicators according to the findings. The format of the LOOKUP statement is

(condition) LOOKUP table, key, index

where **table** is the name of the sequential table to be searched; **key** is a field name, constant, or literal compared with the values in the table entry key fields; and **index** is the name of an implied or numeric field that is set equal to the implied subscript found (see below).

The implied subscript for the table and the (optional) index are set to reference the first entry having a **key** equal to or greater than that of the one specified in the LOOKUP statement. If there is no such entry in the table, the implied subscript and index are set to reference the next available entry address.

The LOOKUP statement sets the #E, #G, and #L indicators as follows:

- a. If a match is found between the specified key and that of a table entry, the #E (equal) indicator is turned on and the other two are turned off.
- b. If no match is found but there is a table entry having a key field greater than the specified key, the #G (greater than) indicator is turned on and the other two turned off.
- c. In other cases, the #L (less than) indicator is turned on and the other two turned off. This is always the case for empty tables. For full tables, the implied subscript is set to the maximum table entry plus one.

## statements

The LOOKUP statement is not applicable to LIFO (last-in-first-out) tables.

### Examples:

```
LOOKUP TABLEA, INPUT.DEPT  
(#E) LOOKUP CTROL, MONTH.IND
```

## CALL STATEMENT

This statement calls a DAS-coded subroutine (appendix E). It has the format

*(condition)* **CALL** *subroutine,argument,...*

where **subroutine** is the name of the DAS-coded subroutine being called, and *argument* is a constant or the name of a record, nonsubscripted field, or table. More than one argument can be included by entering additional arguments, separated by commas, after the first.

The subroutine is usually written in assembler language to perform a task not done directly by the RPG IV program, e.g., special hardware functions or complicated mathematical functions. The incorporation of such subroutines into an RPG IV program is explained in appendix E.

### Examples:

```
(#OV) CALL SQRT, INPVAL, OUTVAL  
(#") CALL CLOCK, TIME  
" SQRT, XXX
```

## READ CARD STATEMENT

This statement reads a card and inputs the data on the card to each of the specified records whose acceptance criteria are met. It has the format

*(condition)* **READ CARD** *name,name,...*

where **name** is the name of the record to receive the data provided its acceptance criteria are satisfied. The data can be read into more than one record by entering additional record names, separated by commas, after the first **name**.

After the card is read, each record specified accepts or rejects the data on the basis of its own selection criteria. If the data are accepted by a record, it enters the record without editing and truncates the right end of the card image if the record is too small to hold all the data.

Control break checks, sequence checks, and auditing checks are performed as required by the record field statements.

If the card read is the last data card (columns 1 and 2 contain /\*), the card image is not read into the records. The #LC indicator and all control break indicators associated with the specified records turn on.

#### Examples:

```
( #25 )      READ CARD EMPLYE
( # " )      READ CARD MAN,WOMAN,CHILD
```

## PRINT STATEMENT

This statement performs page control functions or prints data or messages on the line printer. It has the format

(condition) **PRINT** parameter,parameter,...

where **parameter** is one of the following and the line printer performs the function indicated for the parameter:

|                    |           |                                                                                                                                                                                                                                                                                                                                                  |
|--------------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Record name</b> |           | The record line is printed, the paper advanced to the next line position and the record fields having a B parameter (if any) are cleared. If the record line is longer than the printer line, the rightmost characters of the record are lost. If the bottom of the page is reached by the advancement of the paper, the #OV indicator comes on. |
| <b>\$An</b>        | (n = 1-7) | The paper advances n lines. If the bottom of the page is reached by the advancement of the paper, the #OV indicator comes on. If n is omitted or is zero, the paper advances one line.                                                                                                                                                           |
| <b>\$Cn</b>        | (n = 1-7) | The paper advances to the designated line position on the page as determined by the vertical-format tape in the line printer. If N = 1, the paper advances to the top of the next page and the #OV indicator goes off. If N = 7, the paper advances to the position determined by channel 7 of the tape and the #OV indicator comes on.          |



## statements

Additional parameters can be specified in the same PRINT statement by entering the parameters, separated by commas, after the first **parameter**.

### Examples:

```
(#OV)    PRINT $C1,HEADER,$A1
          PRINT DETAIL
```

## PUNCH STATEMENT

This statement punches one or more cards. It has the format

(condition) **PUNCH** *name*,*name*,...

where *name* is the name of the record to be punched. If the record contains more than 80 characters, the remaining characters are truncated. More than one card can be punched with a single PUNCH statement by entering additional record names, separated by commas, after the first **name**.

### Examples:

```
(KEY=3)  PUNCH SUMMARY
          PUNCH DATA, SUMMRY,H44
```

## STOP STATEMENT

This statement stops the execution of the program and outputs a message to the computer operator. If the operator presses RUN after a STOP, the program continues execution with the statement following the STOP. The STOP statement has the format

(condition) **STOP** *message*

where *message* is the alphanumeric field or literal output to the computer operator.

### Examples:

```
          STOP 'END OF RUN'
(#C1)    STOP 'OUT OF DATA'
(#LC)    STOP
```

## END STATEMENT

This statement is always the last procedural statement, and, therefore, the last statement in the RPG IV program. It is not executable and serves only to indicate the end of the program. This statement has only one form:

**END**

## SECTION 4 — OPERATING PROCEDURES

This section explains how to run the RPG IV programs written according to the instructions in Section 3. The program is first compiled (page 4-1) and the resulting object deck then used to process the data (page 4-5).

### NOTE

In this section, numbers beginning with a zero are octal and numbers beginning with any other digit are decimal. References to the instruction register denote the I register on the 620/f computer and the U register on other members of the Varian 620 computer family. Similarly, START refers to the 620/f and RUN to the others; and RESET on the 620/f is equivalent to SYSTEM RESET on the others. Refer to the applicable system reference manual for descriptions of the control panel switches and indicators.

## COMPILING AN RPG IV PROGRAM

The supplied **RPG IV compiler** card deck is divided into two parts, **COMPILER PART I** and **COMPILER PART II**, each printed on cards of a distinctive color. This compiler accepts RPG IV source (program) statements and, in one pass, produces both a listing of the program and an **object deck**. The listing includes diagnostics and error messages.

## PREPARATION OF THE DECK FOR COMPILEATION

The card deck for compilation is shown in figure 4-1 and comprises, in order:

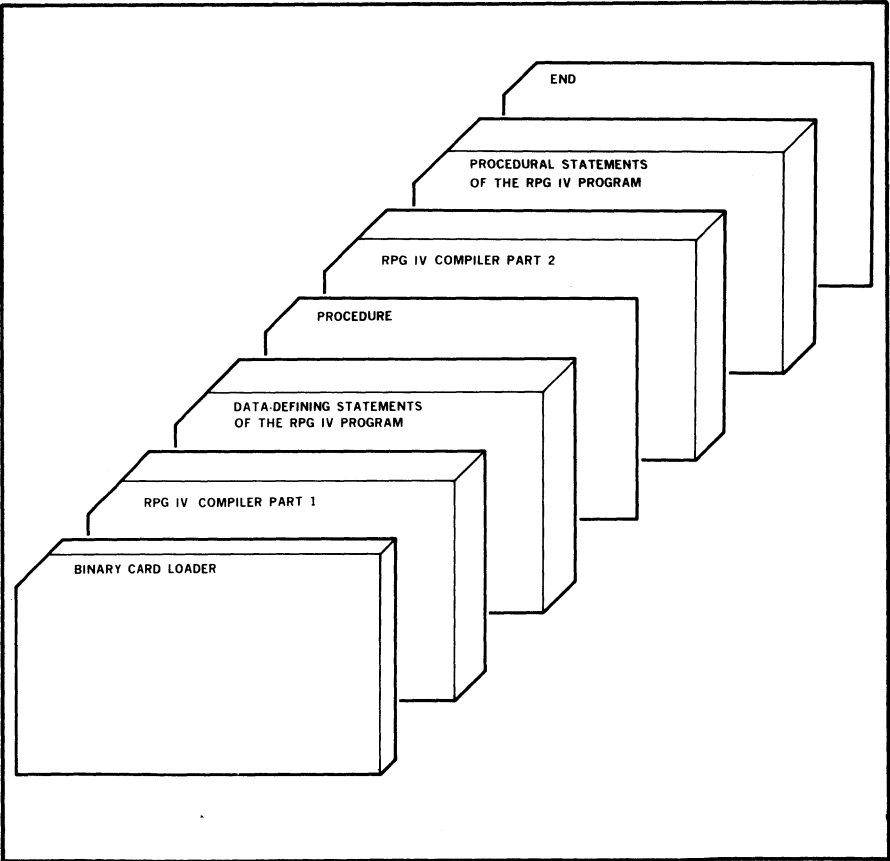
- a. Binary card loader (three cards, supplied)
- b. RPG IV COMPILER PART I
- c. The data-defining statements of the RPG IV program
- d. The PROCEDURE statement of the RPG IV program
- e. RPG IV COMPILER PART II
- f. The procedural statements of the RPG IV program
- g. The END statement of the RPG IV program

The binary card loader is loaded by the card bootstrap loader (appendix D). The binary card loader then loads the RPG IV COMPILER PART I, which then processes the data-defining statements of the program. When this is completed, the binary card loader loads the RPG IV COMPILER PART II, which then processes the procedural statements of the program.

These processes yield a printed listing of the program and an object deck of the compiled program.

**OPERATING THE HARDWARE FOR RPG IV COMPILATION**

To compile a program in RPG IV:



**Figure 4-1. Deck for Compiling an RPG IV Program**

- a. Place the compilation deck (figure 4-1) in the input hopper of the card reader with the binary card loader at the front of the deck.
- b. Turn on and ready the card reader.
- c. Turn on and ready the line printer.
- d. Turn on and ready the card punch. Ensure that there are blank cards in the hopper. If the visual punch station is empty, insert a card into it as follows:
  - (1) Place a card in the auxiliary feed slot.
  - (2) Clear all registers.
  - (3) Set the instruction register to 0100131.
  - (4) Set REPEAT.
  - (5) Press STEP. The card should move from the auxiliary feed slot to the visual punch station.
  - (6) Reset REPEAT.
- e. If it has not already been done, key in the 19-word card bootstrap loader (appendix D). Once done, this step can be omitted.
- f. Clear the A, B, X, and instruction registers.
- g. Set the P register to 000130.
- h. Press RESET or SYSTEM RESET.
- i. Press START or RUN. The cards should start to move through the card reader.

If the compilation is successful, no further manual operation is required. When the END statement is reached, the computer stops in STEP mode with the A, B, and X registers cleared and the P register set to 000130.

Remove the object deck from the output hopper of the card punch and the program listing from the line printer.

## COMPILATION ERRORS

Any of the compilation errors given on page 4-4 stops the compilation. Correction requires the procedure indicated as the recovery from the error. Less serious errors do not stop the compilation but produce an error message as described on the next page and appendix C.

## Compilation Error Halts

Any of the following conditions stops compilation. To correct the error, use the recovery procedure indicated.

**Card reader malfunction:** This is indicated by an instruction register value of 000007. To recover:

- a. Press the START button on the card reader.
- b. Press RESET or SYSTEM RESET on the computer.
- c. Press START or RUN on the computer.

**Card punch malfunction:** This is indicated by an instruction register value of 000031. To recover:

- a. Add cards to the card hopper if it is empty.
- b. If the visual punch station is empty, put a blank card in the auxiliary feed slot and set SENSE switch 1 on the computer.
- c. Press RESET or SYSTEM RESET on the computer.
- d. Press START or RUN on the computer.

**END card found before the PROCEDURE card:** This is indicated by an instruction register value of 0131 and 01 in each of the A, B, and X registers. To recover, remove the compilation deck, reassemble it correctly, and restart.

**Program requires more memory than available:** This is indicated by an instruction register value of 000131 and 0177777 (i.e., - 1) in each of the A, B, and X registers. There is no recovery. Run the program on a system having more memory, or restructure the program into smaller segments.

**Excessive table size:** This error is indicated by instruction register value of 00144. There is no recovery. The user should investigate alternatives for reducing his tables to an acceptable size for his computer memory.

## Compilation of Error Messages

Any of the conditions tabulated in appendix C causes an error message to be printed on the program listing. In addition, there is an arrow pointing to the location of the error as a diagnostic aid. An error message and arrow are shown as follows:

( 1 , 4 ' PAGE '  
↓

SYNTAX

Compilation continues so that the program listing is complete. Thus, all listed errors can be detected and corrected on one pass.

To recover from these errors, correct the program statements containing the errors and recompile the program. Discard any object deck produced from a compilation containing errors.

## PROCESSING DATA WITH A COMPILED RPG IV PROGRAM

The software supplied to run the compiled program is divided into two parts, the **RPG IV loader** and the **RPG IV runtime support program**, each printed on cards of a distinctive color. This software, together with the **object deck** produced by the method given on page 4-1, processes the **data** and generates the completed report.

### PREPARATION OF THE DECK FOR PROCESSING DATA

The card deck for processing data (**runtime deck**) is shown in figure 4-3 and comprises in order:

- a. Binary card loader (the same three cards used in the deck for compilation, (page 4-1)
- b. RPG IV loader
- c. The compiled object deck resulting from the compilation
- d. RPG IV runtime support program
- e. Data cards
- f. Last card (has /\* in columns 1 and 2)

Successful execution (see below) of the program results in the generation of reports on the line printer and/or files on the card punch.

### OPERATING THE HARDWARE FOR PROCESSING RPG IV DATA

To process data in RPG IV, follow the directions given for compilation replacing the compilation deck with the runtime deck (figure 4-2). Successful execution of the program generates the required report on the line printer.

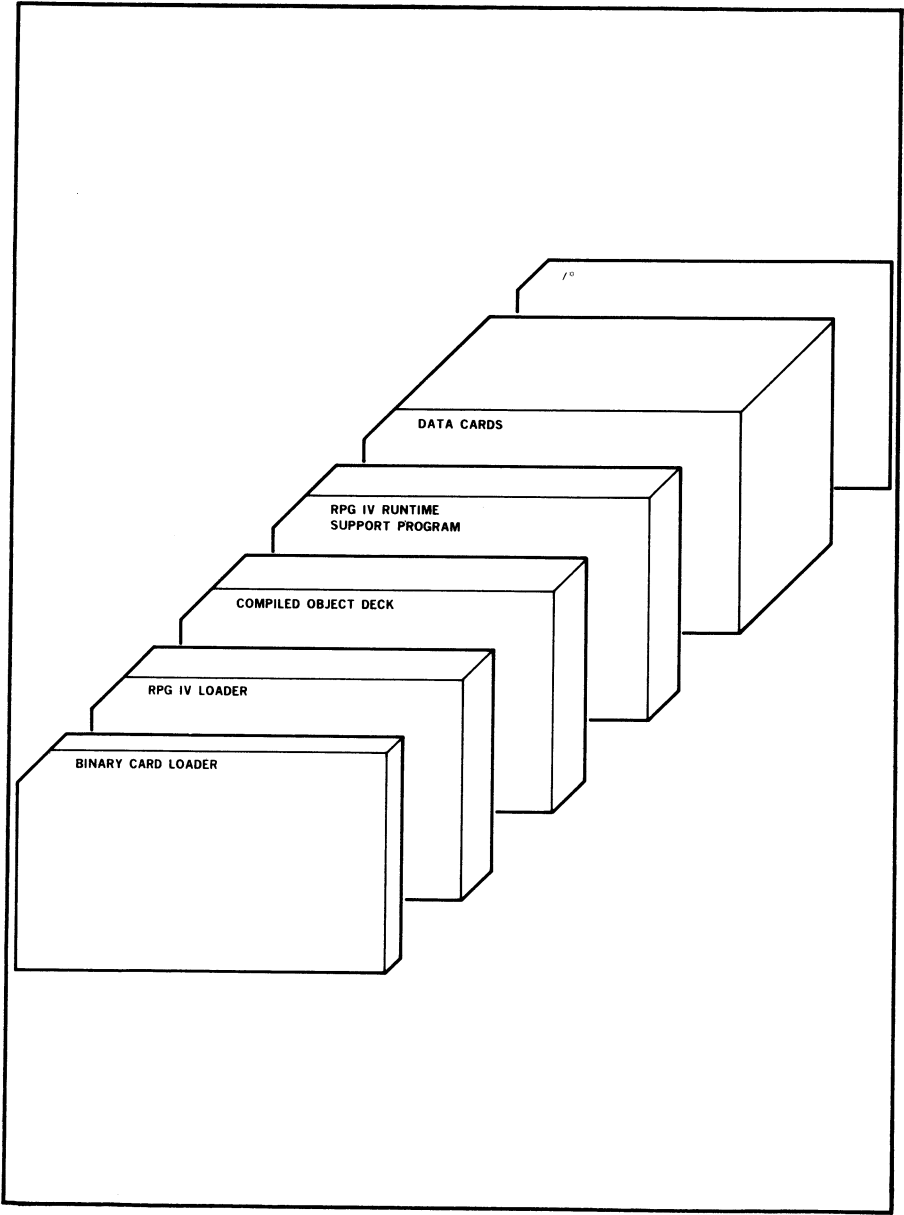


Figure 4-2. Runtime Deck for Processing RPG IV Data

## Loading Errors

Any of the following error messages can appear on the line printer output upon detection of the corresponding loading error. Except for the MISSING PROGRAMS error, any of these errors causes a halt in the loading after the message is printed.

**PROG TOO BIG:** This message indicates that the object deck requires more memory than is available in the system. There is no recovery. Run the program on a system having more memory, or restructure the program into smaller segments.

**INVALID OBJECT DECK:** This message indicates that the loader encountered a directive or format not conforming to those of a normal compiler output. There is no recovery. Recompile the program. If the error persists, check the card punch and recompile.

**CHECKSUM ERROR:** This message indicates that the last card read produced a checksum error. Back up the erroneous card and press READ to reread it. If the error persists, check the card punch and recompile.

**CARD SEQUENCE ERROR:** This message indicates that the cards in the object deck are out of sequence. Correct the sequence and reload. (The eight-bit sequence field on the object deck cards is in rows 6 through 9 of column 1 and rows 12 through 1 of column 2, where row 1 of column 2 is the least significant bit.)

**PROG NOT EXECUTABLE:** This message indicates that the program contains an error that would prevent output of the correct results. There is no recovery. Correct the program and recompile.

**MISSING PROGRAMS:** This message, followed by the names of the programs that are missing, indicates that there is a CALL statement reference to a subprogram not included in the runtime deck. Loading continues and the object program is executed up to the call to the missing program. At this point execution halts. There is no recovery. Insert the missing program in the program (appendix E) and recompile.

## Execution (Runtime) Errors

Any of the following conditions stops execution of the object program. To correct the error, use the recovery procedure indicated.

**STOP statement encountered:** This is indicated by an instruction register value of pf

000001 and an operator message on the line printer. To continue execution, press START or RUN on the computer.



**Missing CALL subroutine:** This is indicated by an instruction register value of 000002. There is no recovery. Insert the missing subroutine in the program deck and recompile.

**Worklist overflow:** This is indicated by an instruction register value of 000003. This results from an overflow on an internal worklist stack during complicated arithmetical manipulations, etc. There is no recovery from this rare condition. Run the program on a system having more memory, or recode the lengthy expression into smaller subexpressions.

**Card reader malfunction:** This is indicated by an instruction register value of 000007. To recover:

- a. Press the START button on the card reader.
- b. Press RESET or SYSTEM RESET on the computer.
- c. Press START or RUN on the computer.

**Card punch malfunction:** This is indicated by an instruction register value of 000031. To recover:

- a. Add cards to the card hopper if it is empty.
- b. If the visual punch station is empty, put a blank card in the auxiliary feed slot and set SENSE switch 1 on the computer.
- c. Press RESET or SYSTEM RESET on the computer.
- d. Press START or RUN on the computer.

## **SECTION 5 – SAMPLE RPG IV PROGRAM**

### **Calendar Program**

This program will print a calendar for any year between 1582 (when the Gregorian calendar was adopted) and 4901. The year 4901 is a terminal date for the Gregorian calendar because by that year the astronomical year will be out of step by one day (this occurs approximately every 3,000 years).

### Note

A year is a leap year if:

- a. It is evenly divisible by 4, but not by 100  
1900 is not; 1904 is.  
or
- b. It is evenly divisible by 400  
1900 is not; 2000 is.

The day of the week can be calculated by use of the formula:

$$DW = ([2.6 * M] + DM + B + [B/4] + [C/4] - 2C) \text{MOD} 7$$

where

DW = Day of the week (0 = Sun., ..., 6 = Sat.)

M = Month (Jan. = 11, Feb. = 12 of previous year and Mar. = 1, ..., Dec. = 10 of current year)

DM = Day of the month (1, ..., 31)

B = Last two digits of the year

C = First two digits of the year

( ) MOD 7 = Remainder after dividing by 7 until ( ) is less than 7

The months are printed in four rows of three columns. The program initialization (lines 10 through 30) inputs the heading cards. Loop 1 (lines 40 through STOP) reads in a year, prints the calendar for that year, and, on reading the last card (/ \*), prints 'END OF JOB' and then stops.

Loop 2 (lines 80 through 190) is inside loop 1, and prints four rows of three months each (rows numbered 0 through 3). Inside loop 2 is loop 3 (lines 110 through 190) which prints six lines (one for each week, see Jan. 1971). Loop 4 (lines 130 through 190), inside loop 3, creates the line of the week for each column (numbered 1, 2, 3). The last loop is 5 (lines 150 through 190) which calculates the day of the week for each day of the month and inserts it into the line. If the day is not a Saturday, the line is shifted left one position, and the next day is tried.

This method has a very slow execution rate, but it demonstrates the use of loops within loops, and of overlapping fields (SO, SP). The figures that follow illustrate this program. Figure 5-1 is the program, 5-2 is the flowchart, 5-3 is the input data, and figure 5-4 is the printed output. [

## VARIAN RPG IV SOURCE LISTING

```

TABLE MD      (5)
      HDNG     (1,72)
RECORD IN
      YIN      (1,4)
      YEAR     (1,4.0)
      C        (1,2.0)
      B        (3,4.0)
      M        (5,6.0)
      DM       (7,9.0)
      SO       (10,12.0)
      SP       (10,13.1)
      SQ       (14,15.0)
RECORD LINE
      YOUT     (35,38),B
      COL1     ( 1,71),B
      COL2     (26,71),B
      COL3     (51,71),B
      COL4     (54,71),B
      O        (70,71.0),B,Z
TABLE T       (12)
      LM       (1,2.0)
      DM       (3,4.0)
PROCEDURE
10      MOVE 1,I
20      READ CARD LINE
      MOVE COL1,MONG(I)
      COMPUTE I=I+1
      (I<6) GO TO 20
      MOVE ' ',COL1
30      MOVE 31,LM(1),LM(3),LM(5),LM(7),LM(8),LM(10),LM(12)
      MOVE 30,LM(4),LM(6),LM(9),LM(11)
40      READ CARD IN
      (WLC) PRINT SC1
      (W") STOP 'END OF JOB'
50      MOVE YIN, YOUT
      PRINT SC1,LINE
60      COMPUTE SP=B/4
      "      SQ=C/4
      MOVE 28,LM(2)
      ((B=4*SP) AND ((B>40) OR (C=4*SQ)))MOVE 29,LM(2)
70      COMPUTE ROW=1
80      COMPUTE ROW=ROW+1
      (ROW>3) GO TO 40

```

Figure 5-1. Calendar Program (1 of 2)

## VARIAN RPG IV SOURCE LISTING

```

90          COMPUTE ROW1=ROW+1
           MOVE MDNG(ROW1),COL1
           PRINT S43,LINE
           MOVE MDNG(5),COL1
           PRINT LINE
100         MOVE 0,DM(1),DM(2),DM(3),WK
110         COMPUTE WK=WK+1
           (WK>6) GO TO R0
120         MOVE 1,COL
130         COMPUTE M=(3*ROW)+COL
140         MOVE COL2,COL1
135         (M=1) COMPUTE YEAR=YEAR-1
           (M=3) "    YEAR=YEAR+1
150         MOVE COL4,COL3
160         COMPUTE DM(COL)=DM(COL)+1
170         COMPUTE N=M-2
           (N<1) "    N=N+12
               "    SP=2.6*N-0.2+DM(COL)+8-2*C+105
               "    SP=80+B/4
               "    DW=80+C/4
               "    SP=DW/7
               "    DW=DW-7*80
           COMPUTE SQ=LH(M)=DM(COL)+1
180         (SQ>0) MOVE DM(COL),D
190         (DW<6) GO TO 150
           COMPUTE COL=COL+1
           (COL<=3) GO TO 130
           PRINT LINE
           GO TO 110
END

```

Figure 5-1. Calendar Program (2 of 2)

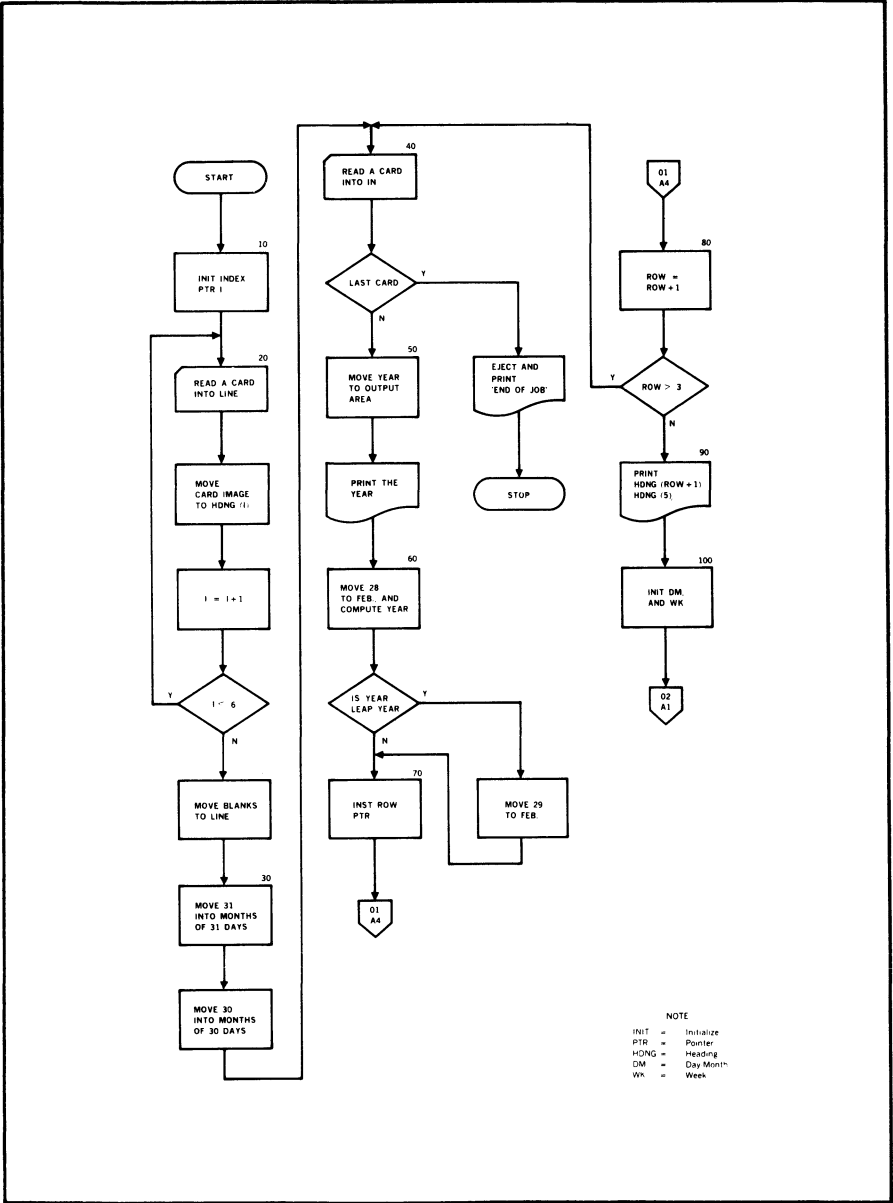


Figure 5-2. Flowchart for Calendar Program (1 of 2)

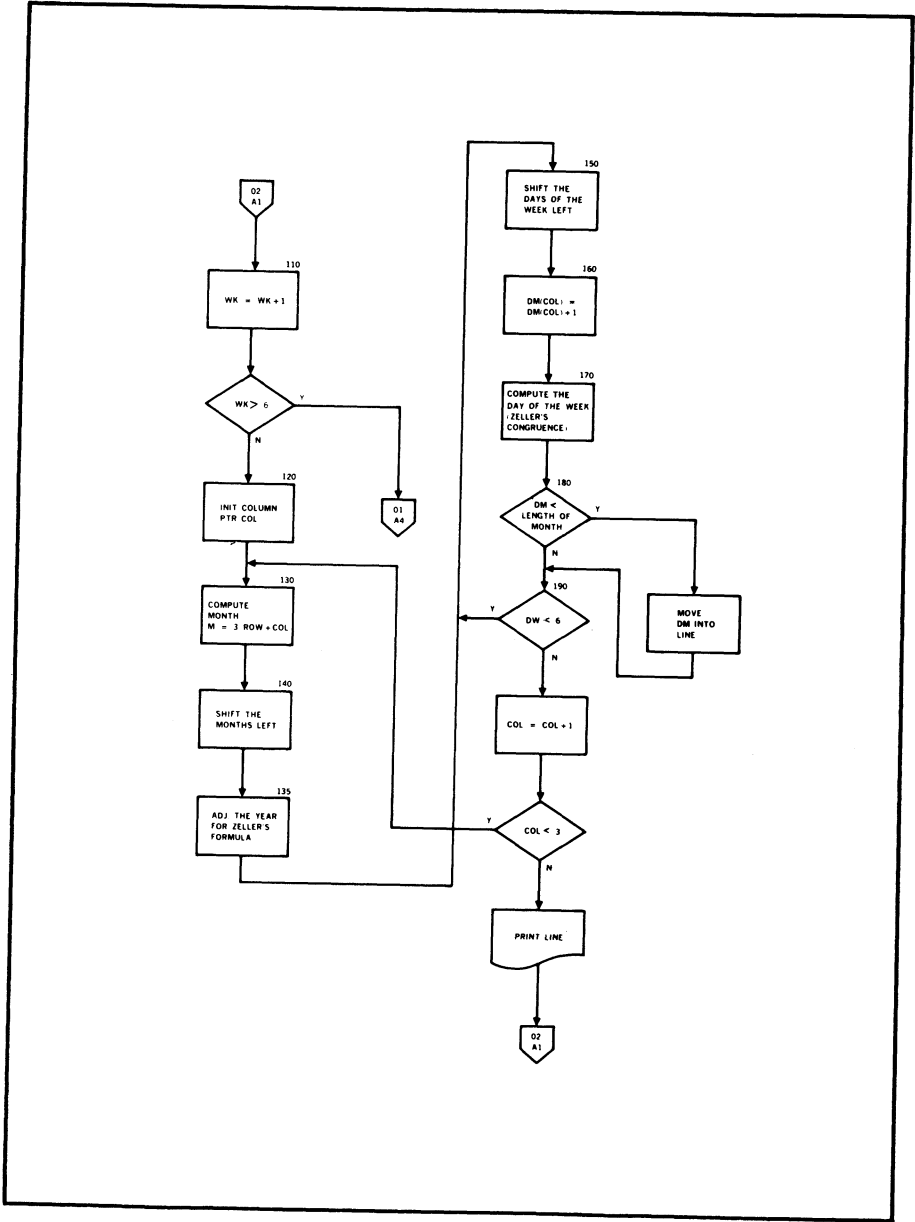


Figure 5-2. Flowchart for Calendar Program (2 of 2)

|               |               |               |   |
|---------------|---------------|---------------|---|
| JANUARY       | FEBRUARY      | MARCH         | 1 |
| APRIL         | MAY           | JUNE          | 2 |
| JULY          | AUGUST        | SEPTEMBER     | 3 |
| OCTOBER       | NOVEMBER      | DECEMBER      | 4 |
| S M T W T F S | S M T W T F S | S M T W T F S | S |
| 1971          |               |               |   |

Figure 5-3. Input Data

1971

JANUARY

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 |    |    |    |    |    |    |

FEBRUARY

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    | 1  | 2  | 3  | 4  | 5  | 6  |
| 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 |    |    |    |    |    |    |

MARCH

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    | 1  | 2  | 3  | 4  | 5  | 6  |
| 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |    |    |    |

APRIL

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 4  | 5  | 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |    |

MAY

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    |    | 1  |
| 2  | 3  | 4  | 5  | 6  | 7  | 8  |
| 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 |    |    |    |    |    |

JUNE

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 |    |    |    |

JULY

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 4  | 5  | 6  | 7  | 8  | 9  | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 | 31 |

AUGUST

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    | 1  | 2  | 3  | 4  | 5  | 6  |
| 8  | 9  | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 |    |    |    |    |

SEPTEMBER

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 |    |    |

OCTOBER

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 |    |    |    |    |    |    |

NOVEMBER

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    | 1  | 2  | 3  | 4  | 5  | 6  |
| 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 |    |    |    |    |

DECEMBER

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| S  | M  | T  | W  | T  | F  | S  |
|    |    |    |    |    | 1  | 2  |
| 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 |    |

END OF JOB

Figure 5-4. Printed Output





## APPENDICES

|                                     | Page |
|-------------------------------------|------|
| A – Indicator Chart                 | A-2  |
| B – Collating Sequence and Codes    | A-4  |
| C – Compilation Error Messages      | A-6  |
| D – Card Bootstrap Loader           | A-11 |
| E – Call Statement Subroutine Usage | A-12 |

INDICATOR CHART

| Where Used                        | Turned on by:                | Turned off by:       |
|-----------------------------------|------------------------------|----------------------|
| GENERAL INDICATORS #1 THROUGH #99 |                              |                      |
| General purpose                   | SET statement                | SET statement        |
| Sequence-checking                 | Fields out of sequence       | Fields in sequence   |
| Auditing                          | Successful audit             | Unsuccessful audit   |
| Record selection                  | Meeting criteria             | Not meeting criteria |
| Table overflow                    | Out-of-range table reference | SET statement        |

CONTROL BREAK INDICATORS #C1 THROUGH #C10

|                |                                                                                            |                                                                                                                                              |
|----------------|--------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Control breaks | Direct updating that changes this or any higher-level control field (except when #F is on) | Direct updating that does not change this control field or direct updating that changes this or any higher-level control field when #F is on |
|----------------|--------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|

SPECIAL INDICATORS

|                                                      |                                 |                                    |
|------------------------------------------------------|---------------------------------|------------------------------------|
| Repeat verb or condition of previous statement<br>#" | Execution of previous statement | Nonexecution of previous statement |
| Where Used                                           | Turned on by:                   | Turned off by:                     |
| Table-searching<br>#G, #L, #E                        | LOOKUP statement                | LOOKUP statement                   |
| Result of computation<br>#P, #Z, #M                  | COMPUTE statement               | COMPUTE statement                  |

|                               |                                                                                                                                                                                     |                                                         |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| First control break<br>#F     | First direct updating<br>of any control field                                                                                                                                       | Subsequent direct up-<br>dating of any control<br>field |
| Computational Overflow<br>#X1 | Numeric value placed<br>in a field that cannot<br>hold it or any com-<br>putation (including<br>subscript and rela-<br>tional expressions)<br>whose value exceeds<br>99,999,999,999 | SET statement                                           |
| Mode error<br>#X2             | Attempted use of a<br>field of wrong mode<br>(numeric/alphanumeric)                                                                                                                 | SET statement                                           |
| Last card<br>#LC              | READ CARD statement<br>after /* card                                                                                                                                                | SET statement                                           |
| Page overflow<br>#OV          | PRINT statement in<br>which the line count<br>= 44 or one that<br>causes a skip to<br>channel 7                                                                                     | PRINT statement that<br>causes a skip to<br>channel 1   |

#### NOTE

The initial state of #OV is on; that of all other indicators is *off*.

## COLLATING SEQUENCE AND CODES

| Sequence | Character | EBCDIC | ASCII | IBM 029 Keypunch |
|----------|-----------|--------|-------|------------------|
| 1        | (blank)   | 64     | 240   |                  |
| 2        | ø         | 74     | 333   | 12-2-8           |
| 3        | •         | 75     | 256   | 12-3-8           |
| 4        | <         | 76     | 274   | 12-4-8           |
| 5        | (         | 77     | 250   | 12-5-8           |
| 6        | +         | 78     | 253   | 12-6-8           |
| 7        | †         | 79     | 336   | 12-7-8           |
| 8        | &         | 80     | 246   | 12               |
| 9        |           | 90     | 241   | 11-2-8           |
| 10       | \$        | 91     | 244   | 11-3-8           |
| 11       | *         | 92     | 252   | 11-4-8           |
| 12       | )         | 93     | 251   | 11-5-8           |
| 13       | ;         | 94     | 273   | 11-6-8           |
| 14       |           | 95     | 334   | 11-7-8           |
| 15       | —         | 96     | 255   | 11               |
| 16       | /         | 97     | 257   | 0-1              |
| 17       | ,         | 107    | 254   | 0-3-8            |
| 18       |           | 108    | 245   | 0-4-8            |
| 19       | —         | 109    | 337   | 0-5-8            |
| 20       | >         | 110    | 276   | 0-6-8            |
| 21       |           | 111    | 277   | 0-7-8            |
| 22       | :         | 122    | 272   | 2-8              |
| 23       | #         | 123    | 243   | 3-8              |
| 24       | @         | 124    | 300   | 4-8              |
| 25       | '         | 125    | 247   | 5-8              |
| 26       | =         | 126    | 275   | 6-8              |
| 27       | "         | 127    | 242   | 7-8              |
| 28       | A         | 193    | 301   | 12-1             |
| 29       | B         | 194    | 302   | 12-2             |
| 30       | C         | 195    | 303   | 12-3             |
| 31       | D         | 196    | 304   | 12-4             |
| 32       | E         | 197    | 305   | 12-5             |
| 33       | F         | 198    | 306   | 12-6             |
| 34       | G         | 199    | 307   | 12-7             |
| 35       | H         | 200    | 310   | 12-8             |
| 36       | I         | 201    | 311   | 12-9             |
| 37       | [         | 208    | 335   | 11-0             |
| 38       | J         | 209    | 312   | 11-1             |
| 39       | K         | 210    | 313   | 11-2             |

| Sequence | Character | EBCDIC | ASCII | IBM 029 Keypunch |
|----------|-----------|--------|-------|------------------|
| 40       | L         | 211    | 314   | 11-3             |
| 41       | M         | 212    | 315   | 11-4             |
| 42       | N         | 213    | 316   | 11-5             |
| 43       | O         | 214    | 317   | 11-6             |
| 44       | P         | 215    | 320   | 11-7             |
| 45       | Q         | 216    | 321   | 11-8             |
| 46       | R         | 217    | 322   | 11-9             |
| 47       | S         | 226    | 323   | 0-2              |
| 48       | T         | 227    | 324   | 0-3              |
| 49       | U         | 228    | 325   | 0-4              |
| 50       | V         | 229    | 326   | 0-5              |
| 51       | W         | 230    | 327   | 0-6              |
| 52       | X         | 231    | 330   | 0-7              |
| 53       | Y         | 232    | 331   | 0-8              |
| 54       | Z         | 233    | 332   | 0-9              |
| 55       | 0         | 240    | 260   | 0                |
| 56       | 1         | 241    | 261   | 1                |
| 57       | 2         | 242    | 262   | 2                |
| 58       | 3         | 243    | 263   | 3                |
| 59       | 4         | 244    | 264   | 4                |
| 60       | 5         | 245    | 265   | 5                |
| 61       | 6         | 246    | 266   | 6                |
| 62       | 7         | 247    | 267   | 7                |
| 63       | 8         | 248    | 270   | 8                |
| 64       | 9         | 249    | 271   | 9                |

## COMPILATION ERROR MESSAGES

### Error

### Location of Arrow

#### Message: INDICATOR

Character other than # at  
beginning of assumed indicator  
symbol

Invalid character

Invalid indicator character

Invalid character

#### Message: INVALID

More than one key field for a  
table

End of second table field state-  
ment containing KEY

DELETE references a LIFO table

Last character of table name

#### Message: LABEL

Duplication of statement numbers

Last digit of statement number

Undefined statement number

None -- message after END

#### Message: LITERAL

No character between opening  
and closing single quotation  
marks

Closing quotation mark

End of line found before closing  
single quotation mark

Last nonblank character

Duplication of table or record  
names

Last character of name

More than six characters in a  
name

Seventh character of name

## COMPILATION ERROR MESSAGES *(continued)*

| Error                                                                           | Location of Arrow              |
|---------------------------------------------------------------------------------|--------------------------------|
| <b>Message: NAME</b>                                                            |                                |
| Field name followed by period or qualified name where nonfield name is required | Period                         |
| Field name of a qualified name not defined in a record or table                 | Last character of field name   |
| Subscripted record or record field name                                         | Left parenthesis of subscript  |
| Invalid reference to name (e.g., a field name where a record name is required)  | Last character of name         |
| LOOKUP references a LIFO table                                                  | Last character of table name   |
| LOOKUP contains a subscript                                                     | Right parenthesis of subscript |
| Subscripted name in CALL argument list                                          | Right parenthesis of subscript |

### **Message: RELATIONAL**

|                                                                                                                       |                                         |
|-----------------------------------------------------------------------------------------------------------------------|-----------------------------------------|
| Relational expression contains two literals, or attempts to compare an alphanumeric field to a constant or expression | Last character of relational expression |
| Zero field boundary                                                                                                   | Last digit of boundary                  |
| Field width not positive                                                                                              | Last digit of field-end spec            |
| Numeric portion of indicator quantity zero or too large                                                               | Last digit of number                    |
| Accumulation of integer quantity causes arithmetic overflow                                                           | Digit causing overflow                  |
| Record selection column zero or > 255                                                                                 | Last digit of selection column          |



**COMPILATION ERROR MESSAGES** *(continued)*

| Error                                | Location of Arrow              |
|--------------------------------------|--------------------------------|
| <b>Message: SIZE</b>                 |                                |
| Statement number zero of > 9999      | Last digit of statement number |
| > 14 digits before decimal point     | Fifteenth digit                |
| > 9 digits after decimal point       | Tenth digit                    |
| Numeric portion of PRINT zero or > 7 | Last digit of number           |

|                                                                          |                                                                                |
|--------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| <b>Message: SYNTAX</b>                                                   |                                                                                |
| First character of name not alphabetic                                   | Invalid character                                                              |
| Missing ( in field definition                                            | Character following name                                                       |
| Missing ) in field definition                                            | Character following last digit of field-end specification or fractional length |
| Missing ) on subscript                                                   | Character following subscript                                                  |
| First character of assumed integer not numeric                           | Invalid character                                                              |
| Field statement found before record or table statement                   | First character of statement                                                   |
| Invalid selection identification                                         | Invalid character                                                              |
| Nonblank character in statement after meaningful specifications complete | Invalid character                                                              |
| Editing character in definition of a nonnumeric field                    | Invalid character                                                              |

## COMPILATION ERROR MESSAGES *(continued)*

|                                                                                  |                                                                 |
|----------------------------------------------------------------------------------|-----------------------------------------------------------------|
| First nonblank character after<br>TABLE in a table statement not (               | Invalid character                                               |
| Missing ) after number of entries<br>in table statement                          | Nonblank character following<br>number of entries               |
| Nonblank character after PROCEDURE                                               | Invalid character                                               |
| Missing ) in conditional expression                                              | Nonblank character following<br>conditional expression          |
| Missing = in COMPUTE statement                                                   | Nonblank character following<br>field name                      |
| Missing indicator list in SET<br>statement                                       | Column 69 of SET statement                                      |
| No condition, ON, or OFF in<br>SET statement                                     | Nonblank character following<br>position meant for missing item |
| Missing ) in indexed GO TO<br>statement                                          | Nonblank character following<br>last statement number in list   |
| Missing to-field in statement<br>that requires one                               | Column 69 of statement                                          |
| Missing index field in LOOKUP<br>statement                                       | Column 69 of LOOKUP statement                                   |
| Missing ) on expression that<br>began with (                                     | Nonblank character following<br>expression                      |
| Missing relational operator after<br>first operand of relational ex-<br>pression | Nonblank character following<br>first operand                   |
| First character of assumed<br>decimal constant neither digit<br>nor period       | Invalid character                                               |
| Missing argument list in PRINT<br>statement                                      | Column 69 of PRINT statement                                    |

**COMPILATION ERROR MESSAGES** *(continued)*

| Error                                                        | Location of Arrow                                 |
|--------------------------------------------------------------|---------------------------------------------------|
| Keyword CARD missing in READ CARD statement                  | First character not matching the sequence C A R D |
| Missing delimiter after READ CARD statement character string | Nonblank character following CARD                 |

## CARD BOOTSTRAP LOADER

Using the data entry switches on the computer front panel, enter the card bootstrap loader as follows:

| Memory Address | Octal Contents |      | DAS Code |           |
|----------------|----------------|------|----------|-----------|
| 000114         | 102530         | BOOR | CIA      | 030       |
| 000115         | 004250         |      | LRLA     | 8         |
| 000116         | 101130         |      | SEN      | 0130,BOOS |
| 000117         | 000122         |      |          |           |
| 000120         | 001000         |      | JMP      | * - 2     |
| 000122         | 102130         | BOOS | INA      | 030       |
| 000123         | 055000         |      | STA      | 0,1       |
| 000124         | 005144         |      | IXR      |           |
| 000125         | 001000         |      | JMP      | BOOU      |
| 000126         | 000131         |      |          |           |
| 000127         | 000000         | BOOT | DATA     | PLD       |
| 000130         | 100230         |      | EXC      | 0230      |
| 000131         | 101130         | BOOU | SEN*     | 0130,BOOT |
| 000132         | 000127         |      |          |           |
| 000133         | 101630         |      | SEN      | 0630,BOOR |
| 000134         | 100114         |      |          |           |
| 000135         | 001000         |      | JMP      | * - 4     |
| 000136         | 000131         |      |          |           |

Clear the registers.

Load the B register with the upper boundary of a 4K memory module, thus delimiting the virtual memory for a given run; e.g., set the B register to 010000, 020000, etc., to indicate a virtual memory of 4K, 8K, etc., words. If the B register contains zero, the loader locates itself at the top of the memory starting with address 0x7660 (x = 0 for a 4K memory, 1 for an 8K memory, etc.).

Set the P register to 000130.

Put the RPG IV loader, followed by the compiler or runtime deck in the card reader.

Press SYSTEM RESET or RESET.

Press RUN or START. The bootstrap loader loads the binary card loader and transfers control to it for further loading.

## CALL STATEMENT SUBROUTINE USAGE

The RPG IV loader contains the subroutine call table STAB, which has the following format:

|           |                                         |
|-----------|-----------------------------------------|
| Word 1    | Number of subroutine names in the table |
| Words 2-4 | Name of a subroutine                    |
| Word 5    | Zeros                                   |

with words 2-5 repeated for each subroutine name in the table. STAB is duplicated in the RPG IV runtime support program with the zero word following each subroutine name changed to give the subroutine entry address as follows:

|           |                                         |
|-----------|-----------------------------------------|
| Word 1    | Number of subroutine names in the table |
| Words 2-4 | Name of a subroutine                    |
| Word 5    | Subroutine entry address                |

with words 2-5 repeated for each subroutine name in the table.

**Linkage by loader:** When the loader encounters an external reference in the object deck, it searches STAB for the name. If the name is in the table, a runtime CALL instruction is stored with the operand equal to the entry number of the subroutine name in STAB; the runtime interpreter uses this value to locate the beginning of any specified subroutine. If the name is not in the table, the operand is zero and the error message MISSING SUBROUTINE output on the line printer.

To retrieve arguments from the CALL statement calling sequence, the called subroutine requires the service of the runtime support routine GETR. GETR is called for each argument in the calling sequence. After each return from GETR, the A register contains a pointer to the first (next) argument in the calling sequence. The interpreter location counter is stepped past the argument on each call to GETR so that, after all of the arguments have been retrieved, the location counter is positioned for the interpreter to continue with the next interpretive instruction. The contents of location REND in the runtime program will be negative for the last argument. To properly interface with GETR, a called subroutine must equate GETR with 0006100 and REND with 0000017.

**Coding CALled Subroutines:** A subroutine whose name appears in a CALL statement of an RPG IV program is first coded in DAS assembly language and assembled with either the DAS 4A or DAS 8A assembler. The object output must be on punched cards. The assembly language source contains the following as the last seven statements of the program:

| Card | Operation | Operand                                                                                                                                                                                                       |
|------|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1    | ORG       | Octal address of the first word for this entry in the subroutine call table (STAB) in the RPG IV runtime support program. The octal location value to be used is shown in figure A-2.                         |
| 2    | DATA      | Subroutine name enclosed between apostrophes. The subroutine name (exclusive of apostrophes) must correspond exactly to the name used in the CALL statement of the RPG IV program.                            |
| 3    | DATA      | Octal address of the subroutine entry location.                                                                                                                                                               |
| 4    | ORG       | Octal address of the first word of the corresponding entry in the subroutine call table (STAB) of the RPG IV loader program. The octal location value for the appropriate table entry is shown in figure A-1. |
| 5    | DATA      | Subroutine name enclosed between apostrophes. The subroutine name must correspond exactly to the name used in the CALL statement of the RPG IV program.                                                       |
| 6    | DATA      | Octal address of the subroutine entry location.                                                                                                                                                               |
| 7    | END       |                                                                                                                                                                                                               |

Card 7 is the END statement of the source deck itself. There is no operand. Assembly language subroutines called by an RPG IV program must return control to the RPG IV runtime interpreter routine. Consequently, the last executed instruction must be:

|                  |                                                                                                                                          |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <b>operation</b> | <b>operand</b>                                                                                                                           |
| <b>JMP</b>       | (Octal address of the entrance to the RPG IV interpreter (INT). The octal location value is shown in figure A-2 opposite the label INT.) |

After assembly, the three final object cards contain the following data:

1. Object data from cards 1, 2, and 3 above
2. Object data from cards 4, 5, and 6 above
3. Normal card object end card

Remove the next-to-last card and place it in the front of the last card of the RPG IV loader object deck. Place the remaining object cards of the assembled routine at the end of the RPG IV runtime support deck with the last card of that deck removed.

Subroutines such as these, which are called at run time by an RPG IV program, reside in that part of upper memory not otherwise used by RPG IV. This area is determined empirically:

1. Initialize each word of memory to some specific value using the 620 AID II program and the command:

**la,b,c,**                      (initialize locations a  
                                         through b to c)

2. Load and run the RPG IV program with the subroutine(s) omitted.

When RPG IV attempts to execute a missing subroutine, the computer halts with 000002 in the instruction register. Using the 620 AID II and the command:

**Sa,b,c,**                      (search locations a through b to c)

scan the computer memory for the value to which memory was previously initialized. That block of memory below the resident card binary loader (which is filled with the value to which memory was previously initialized) is then available for containing the subroutine(s) whose name(s) appear in CALL statements in the RPG IV program.

```

*
*          SUBROUTINE CALL TABLE (STAB)
*          THE FOLLOWING TABLE CONTAINS THE REWIND INFORMATION
*          TO ENABLE THE RPG IV LOADER TO LINK TO AN EXTERNAL
*          SUBROUTINE WHICH IS REFERENCED BY A "CALL"
*          STATEMENT. THE TABLE PROVIDES ROOM FOR
*          TWO ENTRIES. EACH ENTRY CONTAINS THE
*          FOLLOWING DATA:
*
*          1. NAME OF SUBROUTINE
*          2. SUBROUTINE ENTRY ADDRESS
*
*          STAB  #DATA  #2          NUMBER OF ENTRIES
*          #DATA  #1          SUBROUTINE NAME- 1ST ENTRY
*
*          #DATA  #0          SUBROUTINE ENTRY ADDRESS
*          #DATA  #1          SUBROUTINE NAME- 2ND ENTRY
*
*          #DATA  #0          SUBROUTINE ENTRY ADDRESS
*
*          (DATA IN THE SUBROUTINE CALL TABLE MUST
*          CORRESPOND IDENTICALLY TO THE DATA
*          CONTAINED IN THE SUBROUTINE CALL
*          TABLE (SIAE) OF THE RPG IV
*          RELATIVE SUPPORT PROGRAM)

```

| STAB   | #DATA  | #2 | NUMBER OF ENTRIES          |
|--------|--------|----|----------------------------|
| 001155 | 000002 |    |                            |
| 001156 | 120240 | #1 | SUBROUTINE NAME- 1ST ENTRY |
| 001157 | 120240 |    |                            |
| 001160 | 120240 |    |                            |
| 001161 | 000000 | #0 | SUBROUTINE ENTRY ADDRESS   |
| 001162 | 120240 | #1 | SUBROUTINE NAME- 2ND ENTRY |
| 001163 | 120240 |    |                            |
| 001164 | 120240 |    |                            |
| 001165 | 000000 | #0 | SUBROUTINE ENTRY ADDRESS   |

Figure A-1.



### Figure A-2

# **Master Operating System (MOS) Reference Manual**





# TABLE OF CONTENTS

## SECTION 1 INTRODUCTION

|                             |     |
|-----------------------------|-----|
| SYSTEM CONFIGURATION .....  | 1-1 |
| MOS COMPONENTS .....        | 1-2 |
| Resident Partition .....    | 1-2 |
| Nonresident Partition ..... | 1-2 |
| BIBLIOGRAPHY .....          | 1-4 |

## SECTION 2 CONTROL DIRECTIVES

|                                        |      |
|----------------------------------------|------|
| TYPES AND FORMATS .....                | 2-1  |
| EXECUTIVE CONTROL DIRECTIVES .....     | 2-2  |
| I/O CONTROL DIRECTIVES .....           | 2-5  |
| SYSTEM LOADER CONTROL DIRECTIVES ..... | 2-7  |
| DECK PREPARATION .....                 | 2-14 |

## SECTION 3 INPUT/OUTPUT CONTROL PROGRAM

|                                  |      |
|----------------------------------|------|
| INTRODUCTION .....               | 3-1  |
| LOGICAL AND PHYSICAL UNITS ..... | 3-1  |
| I/O CALLS .....                  | 3-7  |
| Read Binary Record .....         | 3-9  |
| Read Alphanumeric Record .....   | 3-9  |
| Read BCD Record .....            | 3-9  |
| Write Binary Record .....        | 3-10 |
| Write Alphanumeric Record .....  | 3-10 |
| Write BCD Record .....           | 3-11 |
| Write End of File .....          | 3-11 |
| Rewind .....                     | 3-12 |
| Skip Records Forward .....       | 3-12 |
| Skip Records Reverse .....       | 3-12 |

|                            |      |
|----------------------------|------|
| Skip Files Forward .....   | 3-13 |
| Skip Files Reverse .....   | 3-13 |
| Perform Function .....     | 3-14 |
| Request Status .....       | 3-15 |
| PROGRAMMING EXAMPLES ..... | 3-16 |

## SECTION 4 DEBUGGING PROGRAM

|                                      |     |
|--------------------------------------|-----|
| INTRODUCTION .....                   | 4-1 |
| TELETYPE DIALOG .....                | 4-1 |
| PSEUDOREGISTERS .....                | 4-2 |
| INSTRUCTION LANGUAGE .....           | 4-3 |
| Display and Alter Instructions ..... | 4-3 |
| I/O Instructions .....               | 4-4 |
| Control Instructions .....           | 4-5 |
| EXAMPLES OF DEBUGGING .....          | 4-6 |

## SECTION 5 CONCORDANCE PROGRAM

## SECTION 6 FILE EDITING PROGRAM

|                              |     |
|------------------------------|-----|
| PROGRAM AND DIRECTIVES ..... | 6-1 |
| SOURCE FILE .....            | 6-6 |
| Header Record .....          | 6-6 |
| Data Record .....            | 6-7 |
| Catalog Record .....         | 6-7 |

## SECTION 7 SYSTEM MAINTENANCE PROGRAM

## SECTION 8 SYSTEM PREPARATION PROGRAM

|                          |     |
|--------------------------|-----|
| INTRODUCTION .....       | 8-1 |
| CONTROL DIRECTIVES ..... | 8-2 |

INSTALLATION SYSTEM LIBRARY ORGANIZATION ..... 8-13

System Preparation Section ..... 8-13

System Processor Section ..... 8-13

System Library Section ..... 8-14

OPERATING PROCEDURES ..... 8-15

Loading ..... 8-15

Assignment ..... 8-16

Disc Formatting ..... 8-19

System Verification and Completion ..... 8-19

EXAMPLES ..... 8-24

**SECTION 9**  
**LANGUAGE PROCESSORS**

DAS MR ASSEMBLER ..... 9-1

FORTRAN IV COMPILER ..... 9-2

**SECTION 10**  
**SUPPORT LIBRARY**

CALLING SEQUENCE ..... 10-1

NUMBER TYPES AND FORMATS ..... 10-2

SUBROUTINE DESCRIPTIONS ..... 10-4

**SECTION 11**  
**MOS OPERATING PROCEDURES**

DEVICE INITIALIZATION ..... 11-1

Card Reader ..... 11-1

Card Punch ..... 11-1

33/35 ASR Teletype ..... 11-1

High-Speed Paper Tape Reader ..... 11-2

Magnetic Tape Unit ..... 11-2

Magnetic Drum Unit ..... 11-2

Fixed-Head Disc Unit ..... 11-2

Moving-Head Disc Unit (620-39) ..... 11-2

Moving-Head Disc Unit (620-40, -41) ..... 11-2

BOOTSTRAP ..... 11-3

SYSTEM (RE)INITIALIZATION ..... 11-5

## SECTION 12

### USER-CODED I/O DRIVERS

|                                       |       |
|---------------------------------------|-------|
| DEVICE SPECIFICATION TABLE .....      | 12-2  |
| Word 0 .....                          | 12-3  |
| Word 1 .....                          | 12-3  |
| Words 2 and 3 .....                   | 12-4  |
| Word 4 .....                          | 12-4  |
| Word 5 .....                          | 12-4  |
| Word 6 .....                          | 12-4  |
| Words 7 and 8 .....                   | 12-5  |
| Word 9 .....                          | 12-5  |
| Word 10 .....                         | 12-5  |
| Word 11 .....                         | 12-6  |
| Word 12 .....                         | 12-6  |
| Word 13 .....                         | 12-7  |
| Word 14 .....                         | 12-7  |
| Word 15 .....                         | 12-7  |
| I/O DRIVER PROGRAMMING EXAMPLES ..... | 12-13 |
| I/O SUPPORT SUBROUTINES .....         | 12-16 |
| I/O STATUS MESSAGES .....             | 12-17 |
| BIC CONTROL .....                     | 12-18 |
| BIC Control Table .....               | 12-18 |
| BIR\$ .....                           | 12-19 |
| BIA\$ .....                           | 12-19 |

## SECTION 13

### STATUS AND ERROR MESSAGES

|                                  |       |
|----------------------------------|-------|
| EXECUTIVE .....                  | 13-1  |
| SYSTEM LOADER .....              | 13-3  |
| I/O CONTROL .....                | 13-5  |
| LANGUAGE PROCESSORS .....        | 13-5  |
| DAS MR ASSEMBLER .....           | 13-6  |
| FORTRAN IV COMPILER .....        | 13-8  |
| FILE EDITING PROGRAM .....       | 13-10 |
| SYSTEM MAINTENANCE PROGRAM ..... | 13-12 |

|                          |       |
|--------------------------|-------|
| SYSTEM PREPARATION ..... | 13-14 |
|--------------------------|-------|

## APPENDICES

|                                           |             |
|-------------------------------------------|-------------|
| <b>A – ABSOLUTE MODULE FORMAT .....</b>   | <b>A-1</b>  |
| <b>B – OBJECT MODULE FORMAT .....</b>     | <b>A-3</b>  |
| <b>C – DATA FORMAT .....</b>              | <b>A-11</b> |
| <b>D – STANDARD CHARACTER CODES .....</b> | <b>A-17</b> |
| <b>E – TELETYPE CHARACTER CODES .....</b> | <b>A-20</b> |

## LIST OF FIGURES

|                                               |       |
|-----------------------------------------------|-------|
| 1-1 System Partitions and Flow .....          | 1-3   |
| 2-1 Loader Memory Map .....                   | 2-8   |
| 2-2 Memory Map Format .....                   | 2-9   |
| 2-3 Dump Format .....                         | 2-10  |
| 3-1 MOS I/O Units .....                       | 3-2   |
| 3-2 MOS Physical I/O Devices .....            | 3-3   |
| 3-3 Valid Logical Unit Assignments .....      | 3-5   |
| 3-6 Summary of I/O Calls .....                | 3-4   |
| 5-1 Sample Concordance Listing .....          | 5-2   |
| 6-1 Output Catalog Format .....               | 6-3   |
| 6-2 Audit Listing Format .....                | 6-4   |
| 6-3 MOS Source File Structure .....           | 6-8   |
| 7-1 System Maintenance List Output .....      | 7-4   |
| 8-1 Valid SPP Logical Unit Assignments .....  | 8-11  |
| 8-2 620 Bootstrap Loading Routines .....      | 8-17  |
| 8-3 622 Bootstrap Loading Routines .....      | 8-18  |
| 8-4 Logical Unit Functions .....              | 8-20  |
| 8-5 System Preparation List Output .....      | 8-21  |
| 10-1 DAS Coded Subroutines .....              | 10-5  |
| 10-2 FORTRAN IV Coded Subroutines .....       | 10-11 |
| 11-1 MOS System File Bootstrap Routines ..... | 11-4  |
| 12-1 I/O Drivers and Peripheral Devices ..... | 12-9  |





## **SECTION 1 – INTRODUCTION**

The Master Operating System (MOS) is a batch processing operating system for the Varian 620/622 family of computer systems. MOS operates on a wide range of hardware configurations. It is modular, thus facilitating expansion (e.g., new language processors, special user I/O drivers, etc.). MOS makes optimum use of memory by loading only those portions of the system (including I/O) required during execution. Features of MOS include:

- Minimum operator intervention required
- Single tape, drum, or disc as secondary storage device
- Extensive job control language (22 directives)
- Multisource input during loading
- Debugging aids
- File maintenance and editing programs
- Extensive status- and error-reporting

## **SYSTEM CONFIGURATION**

The minimum MOS configuration requires the following hardware:

- a. Varian 620 or 622 series computer
- b. 4K memory increment module
- c. 33/35 ASR Teletype
- d. One of the following:
  - (1) Magnetic tape unit
  - (2) Rotating memory unit on a buffer interlace controller (BIC)

MOS supports and is enhanced by the following hardware:

## **introduction**

- a. Card reader and/or punch
- b. Line printer
- c. High-speed paper tape reader and/or punch
- d. Memory increment(s)
- e. Hardware multiply/divide and extended addressing

## **MOS COMPONENTS**

MOS is divided into resident and nonresident partitions. Figure 1-1 shows their relationship.

### **RESIDENT PARTITION**

This partition comprises the:

- a. Resident monitor
- b. Absolute loader
- c. I/O assignment tables
- d. System flags and parameters
- e. Dump

### **NONRESIDENT PARTITIONS**

This partition comprises the:

- a. Control programs
- b. Support programs
- c. Language processors

### **Control Programs**

The control programs are the:

- a. Executive - job control processor and system control
- b. System loader - linking and relocating loading of system and user programs
- c. I/O control - dispatching of I/O requests and device driving

## Support Programs

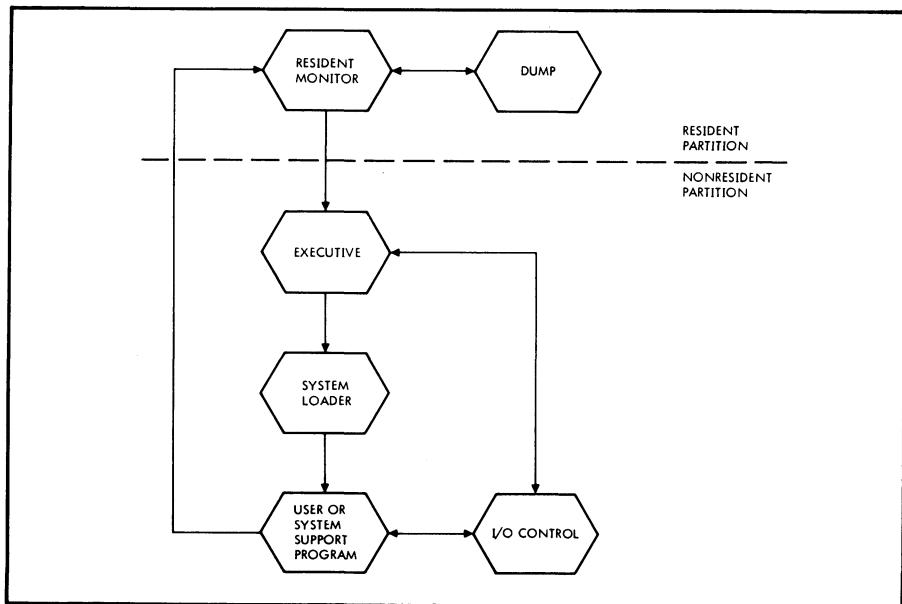
The support programs are:

- a. Math and support library
- b. Concordance program
- c. Debugging program
- d. File editing program
- e. File maintenance program
- f. System preparation program (operates in stand-alone mode)

## Language Processors

The language processors are:

- a. DAS MR assembler
- b. FORTRAN IV compiler (requires an additional memory increment)



**Figure 1-1. System Partitions and Flow**

## BIBLIOGRAPHY

The following gives the stock numbers of manuals pertinent to the use of MOS and the 620 family of computers:

| Title                       | Stock Number  |
|-----------------------------|---------------|
| 620/f Computer Handbook     | 98 A 9908 002 |
| 620/L Computer Handbook     | 98 A 9905 000 |
| 620 Subroutine Descriptions | 98 A 9902 044 |
| 620 Training Manual         | 98 A 9902 503 |
| 620 FORTRAN IV Reference    | 98 A 9902 037 |
| 620/f Maintenance Manual    | 98 A 9908 051 |
| 620/L Maintenance Manual    | 98 A 9905 050 |

Maintenance information is in the following 620 MOS Software Performance Specifications:

| Document No. | Title                                                     |
|--------------|-----------------------------------------------------------|
| 89A0011      | Executive                                                 |
| 89A0021      | System loader                                             |
| 89A0159      | DAS MR assembler                                          |
| 89A0023      | Input/output control system                               |
| 89A0024      | Teletype drivers (620-06, -07, -08)                       |
| 89A0028      | Magnetic tape drivers (620-30, -300, -31, -310)           |
| 89A0030      | Master operating system overview                          |
| 89A0032      | Card reader driver (620-22, -23, -25)                     |
| 89A0033      | High-speed paper tape system driver<br>(620-50, -51, -52) |
| 89A0038      | Debugging program                                         |
| 89A0042      | FORTTRAN IV compiler                                      |
| 89A0048      | System maintenance program                                |
| 89A0049      | MOS system preparation program                            |
| 89A0053      | Source editor                                             |
| 89A0069      | Drum driver (620-38C, -46, -47, -48, -49)                 |
| 89A0094      | Concordance program                                       |
| 89A0102      | Support library                                           |
| 89A0114      | Card punch driver (620-26, -27)                           |
| 89A0115      | Disc driver (620-40, -41)                                 |
| 89A0117      | System preparation (620-40, -41)                          |
| 89A0118      | Line printer driver (620-76, -77)                         |
| 89A0126      | Disc driver (620-39)                                      |
| 89A0127      | System preparation (620-39)                               |
| 89A0130      | MOS format routine (620-40, -41)                          |
| 89A0221      | Disc driver (620-37)                                      |
| 89A0222      | Status 21 printer simulator, driver (620-74)              |
| 89A0223      | System preparation (620-37)                               |

## SECTION 2 – CONTROL DIRECTIVES

### TYPES AND FORMAT

The MOS recognizes three types of control directives:

- a. Executive control directives
- b. I/O control directives
- c. System loader directives

Executive and I/O control directives are executed while the executive is in memory. System loader directives cause the executive to be overlayed with the system program that is implemented by the directive. MOS has the following directives:

| Executive | I/O      | System Loader |
|-----------|----------|---------------|
| JOB       | COPYA    | LOAD          |
| ENDJOB    | COPYB    | ULOAD         |
| ASSIGN    | REW      | ASSEMBLE      |
| IOLIST    | WEOF     | FORTRAN       |
| FORM      | SREC     | SMAIN         |
| STACK     | BREC     |               |
| EOF       | SFILE    |               |
| DATE      | FUNCTION |               |
| COMMENT   |          |               |

The general form of a directive is an alphanumeric record of up to 72 characters in the following format:

**/name,p(1),p(2),p(3),...,p(n)**

where

|                 |                                                                                                                              |
|-----------------|------------------------------------------------------------------------------------------------------------------------------|
| <b>/</b>        | in the first character position of the record specifies that the record contains a control directive.                        |
| <b>name</b>     | is the name of the control directive comprising one to eight alphanumeric characters, and is terminated by a comma or blank. |
| <b>p(1),...</b> | is a parameter string with individual parameters separated by commas.                                                        |

## control directives

The form and number of parameters varies with the directive. However, parameter strings cannot be longer than one record (72 characters). If it is, the directive is truncated.

The parameter string is terminated either by one period or by blanks from the last parameter to character position 72. Blanks within the parameter string are ignored.

Some control directives have an abbreviated form that is equivalent to the full form:

```
/C  
/COMMENT
```

These forms can be used interchangeably.

## EXECUTIVE CONTROL DIRECTIVES

```
/JOB,title
```

This control directive starts a job. (A job is all tasks requested between a /JOB and an /ENDJOB directive.) The system is initialized by setting certain resident constants and logical unit assignments to their default values; a top-of-form function is sent to the list output.

One parameter (optional) is allowed. It is an alphanumeric string of up to eight characters that is stored in the resident monitor. It is an identification printed on every page of list output generated by the assembler or compiler. The identification is also incorporated into the actual object module. If the parameter has fewer than eight characters, the identification is left-justified and filled out with blanks. In the case of a /JOB parameter, the first blank terminates it, i.e., embedded blanks cause truncation.

The parameter can be accessed during execution by referencing the four-word area \$TTL in the resident monitor. This can be done by making \$TTL an external reference. The parameter is stored in ASCII, two characters per word.

```
/ENDJOB
```

This control directive ends a job. The system is initialized by setting certain resident constants and logical unit assignments to their default values; a top-of-form function is sent to the list output. There are no parameters. Any text in the parameter field is ignored.

**/ASSIGN,l(1)=r(1),l(2)=r(2),...,l(n)=r(n)**

This control directive equates and assigns particular logical units to specific physical I/O devices. Execution of this directive decodes the parameter string and alters the logical unit table as specified by the parameter.

The parameters can be logical unit numbers, logical unit names, or physical unit names (figure 3-1). In each parameter pair (i.e., each  $l(n) = r(n)$ ), the left parameter,  $l(n)$ , is a logical unit number or name, and the right parameter,  $r(n)$ , is a logical unit number or name or a physical device name.

In any case, the logical unit to the left of the equal sign is assigned to the unit/device to the right.

If  $r$  is a physical device, the  $l$  entry in the logical unit table is altered so that it points to the physical device driver specified by  $r$ . Thereafter, all I/O operations referencing  $l$  are directed to the physical device specified by  $r$ .

If  $r$  is a logical unit number or name,  $l$  is made equivalent to  $r$  and is assigned to the same physical device as  $r$ . However, if  $r$  is reassigned later to a new physical device,  $l$  no longer has an equivalent assignment.

As many parameter pairs as will fit in the control directive record can be specified on one /ASSIGN. Once a logical unit assignment is made, it remains in effect until changed by a new /ASSIGN, until the system is initialized by /JOB, /ENDJOB, or a bootstrap loading.

**/IOLIST,p(1),p(2),...,p(n)**

This control directive requests a listing of the current assignments of the individual logical units. The parameter string consists of logical unit numbers or names. As many logical units as will fit in the parameter field are allowed. The list is printed on LO and system output (SO, figure 3-1).

#### **Example:**

If the system file is currently assigned to drum unit 0 and the binary output to the high-speed paper tape punch, the directive

**/IOLIST,SF,BO**

prints the following on LO and SO:

**SF = DR00  
BO = PT00**

If the parameter field is blank, all logical units and their assignments are printed, but only on LO.



## control directives

### **/FORM, X**

This control directive sets the value of the line-count word in the resident monitor to the value of parameter x. This word specifies, to operating system programs, the number of lines on the LO file before a top-of-form request is sent to the device. The parameter is a positive decimal integer from 5 to 9999. If the parameter is blank or less than 5, the value is set to the default value of 44.

### **/STACK**

This control directive stacks binary object programs on the BO. Normally, the system rewinds the BO before each assembly or compilation. However, with /STACK in effect, the binary output of all tasks within a job are written on the BO file sequentially.

/STACK remains in effect until a /JOB or /ENDJOB is encountered. It is not set when the MOS is initialized after a bootstrap loading.

There are no parameters. The parameter field is ignored.

### **/EOF**

This control directive instructs the executive to write an end-of-file record on the BO and GO (figure 3-1) files.

There are no parameters. The parameter field is ignored.

### **/DATE, xxxxxxxx**

This control directive inputs the parameter into the operating system. The one parameter is an alphanumeric character string of up to eight characters (for example, month, day, and year). The first blank terminates it, i.e., embedded blanks cause truncation. The parameter is output on any list output generated by the assembler or compiler and other system support programs.

If the parameter field has fewer than eight characters, it is left-justified and filled out with blanks. Once entered, the parameter remains unchanged until another /DATE is encountered.

Access to the parameter during execution is by referencing the four-word area \$DAT in the resident monitor. This can be done by making \$DAT an external reference. The parameter is stored in ASCII, two characters per word.

### **/C**

### **/COMMENT**

This control directive annotates the list output. The MOS prints all 72 characters of this directive record, including /C or /COMMENT, on SO and LO. No other action is taken.

## I/O CONTROL DIRECTIVES

**/COPYA, l(1)=r(1), l(2)=r(2), ..., l(n)=r(n)**

This control directive copies ASCII-coded data files from one logical unit on another. The parameter string comprises logical unit pairs. Copying is record by record from the left (l) unit to the right (r) unit. Copying begins at the position of the l unit when the /COPYA is requested and continues until an end of file is encountered on the l unit. The r unit is not positioned before copying to it begins. After the copy is completed, neither unit is positioned nor is an end of file written on the r unit. Unit l is a logical unit number or name; unit r is a logical unit number or name.

The MOS uses all unused memory as a copying buffer. Therefore, the maximum record length depends on the units.

**/COPYB, l(1)=r(1), l(2)=r(2), ..., l(n)=r(n)**

This control directive is identical to /COPYA, except that copying is binary.

**/REW, p(1), p(2), ..., p(n)**

This control directive rewinds the specified logical units. If a specified unit cannot be rewound, no action is taken for that unit.

**/WEOF, p(1), p(2), ..., p(n)**

This control directive writes end-of-file records on the specified logical units. The format of the end-of-file record depends on the physical device to which the logical unit is currently assigned. If a physical device cannot accept an end of file, no action is taken for that logical unit.

**/SREC, l(1), r(1), l(2), r(2), ..., l(n), r(n)**

This control directive skips physical records on the specified logical units. The parameter string consists of parameter pairs, each pair specifying a logical unit and a record count. /SREC spaces the logical unit forward the number of records designated.

The record count is a positive decimal integer from 0 to 9999. If the count is blank, or if the specified unit cannot skip records, no action is taken for that unit. If the count exceeds 9999, the left four digits are used.

## control directives

**/BREC,l(1),r(1),l(2),r(2),...,l(n),r(n)**

This control directive is identical to /SREC, except that the records on the specified logical unit are skipped in reverse order (backspace).

**/SFILE,l(1),r(1),l(2),r(2),...,l(n),r(n)**

This control directive skips physical files on specified logical units. File-skipping occurs only in the forward direction. The parameter string consists of parameter pairs, each pair specifying a logical unit and a file count. /SFILE spaces the logical unit forward the number of physical files designated.

The file count is a positive decimal integer from 0 to 9999. If the count is blank, or if the specified unit cannot skip files, no action is taken for that unit. If the count exceeds 9999, the left four digits are used.

**/FUNCTION**

**/FUNC,l(1),r(1),l(2),r(2),...,l(n),r(n)**

This control directive performs special functions on specified logical units. The parameter string consists of parameter pairs, each pair specifying a logical unit and a special function code.

The special function code is a positive decimal integer from 0 to 9999. If the code is blank, no action is taken for that unit. If the code exceeds 9999, the left four digits are used.

The function performed depends on the physical device to which the logical unit is currently assigned. Definitions of the I/O functions of MOS are given in Section 3.

## SYSTEM LOADER CONTROL DIRECTIVES

The system loader can load unconditionally from binary input (BI, figure 3-1 ) and/or selectively by program name from SF. It accepts only relocatable object text, including literal addressing and external program-linking. Upon successful completion, the system loader returns control to the resident monitor for program execution. When errors occur during loading, the process is aborted and control returned to the resident monitor, the executive is loaded and the error message posted on the LO and SO. Figures 2-1 and 2-2 show a map of memory during and after the loading process.

```
/LOAD
/L,p(1),p(2),...,p(n)
```

This control directive directs the executive to call the system loader and load one or more object modules from the BI file. The parameter string can specify the following tasks for the loader during loading, or for the resident monitor after the loaded program has been run:

- |              |                                                                                                                                                                                                                                    |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PAUSE</b> | The loader pauses before each program is loaded from BI, allowing the loading of programs from more than one tape by stopping the computer during the changing of tapes.                                                           |
| <b>HALT</b>  | The resident monitor stops after all programs are loaded but before execution begins.                                                                                                                                              |
| <b>MAP</b>   | When loading is complete, the loader outputs a map of all entry points, external names, and labeled data blocks (figure 2-3)                                                                                                       |
| <b>DUMP</b>  | <p>After the loaded program has been executed, the resident monitor dumps core on LO (figure 2-3).</p> <p>The dump routine uses memory locations 0400 through 0477 and thus destroys the original contents of these locations.</p> |
| <b>DEBUG</b> | The loader loads the debugging program as part of the loading task.                                                                                                                                                                |

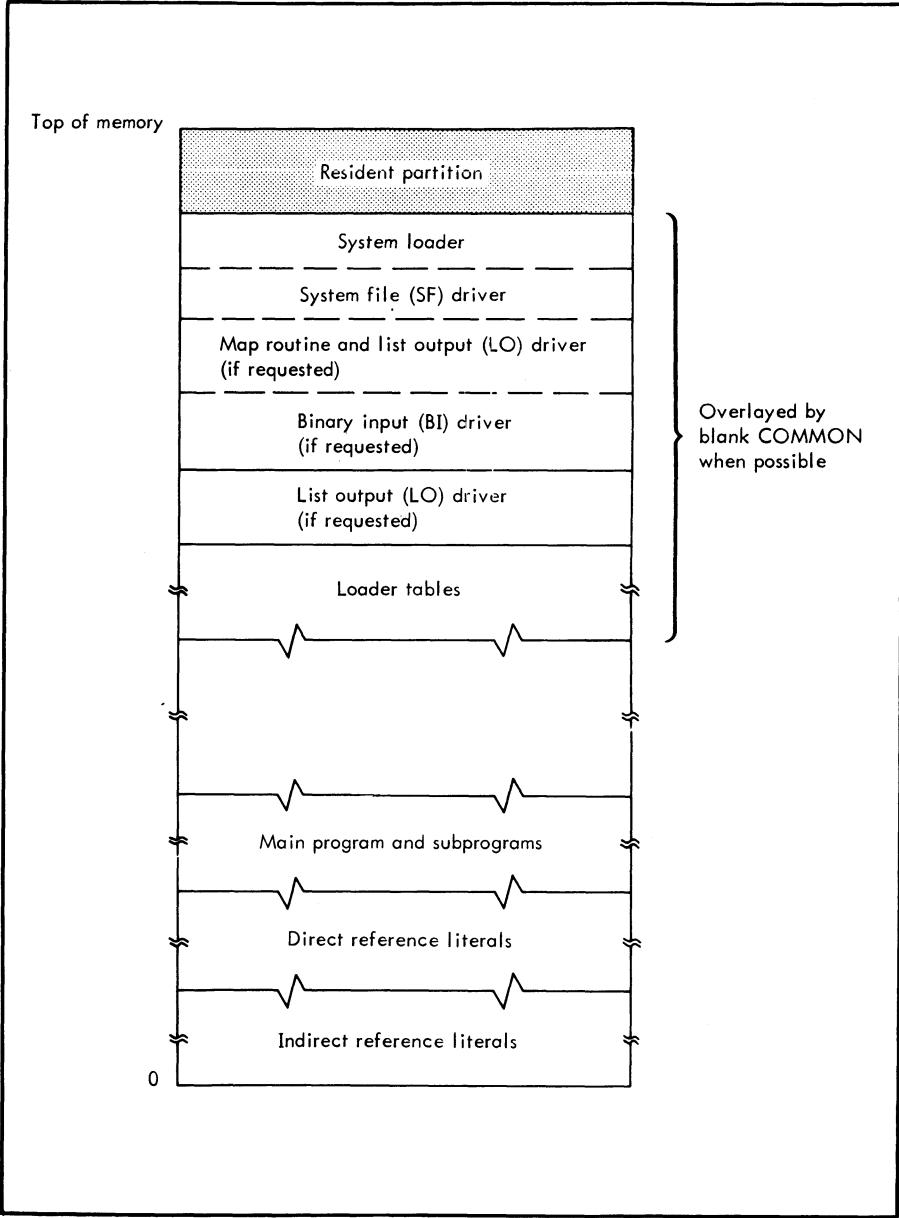


Figure 2-1. Loader Memory Map

THE FOLLOWING IS OUTPUT ON THE LO DEVICE WHEN A MAP REQUEST IS MADE WITH AN ASSEMBLY, COMPILATION, OR LOADING:

|         |       |
|---------|-------|
| ssssss  | aaaaa |
| .       | .     |
| .       | .     |
| ssssss/ | rrrrr |
| .       | .     |
| .       | .     |
| (\$IAP) | vvvvv |
| (\$LIT) | vvvvv |
| (\$PED) | vvvvv |

WHERE

ssssss IS A SYMBOLIC NAME, RIGHT-JUSTIFIED.

aaaaa IS AN OCTAL ENTRY ADDRESS, RIGHT-JUSTIFIED, OF A LOADED PROGRAM OR COMMON BLOCK NAME.

rrrrr IS AN ADDRESS-REFERENCING OF AN UNLOADED PROGRAM OR COMMON BLOCK NAME.

vvvvv IS THE OCTAL VALUE OF:

(\$IAP) HIGHEST INDIRECT ADDRESS POOL LOCATION

(\$LIT) LOWEST DIRECT REFERENCE LITERAL POOL LOCATION

(\$PED) HIGHEST PROGRAM LOCATION STORED

Figure 2-2. Memory Map Format

```

000000 001000 035111 001000 034715 001000 034715 000350 103121
000001 000265 003712 105717 105720 105721 105776 000013 035656
000002 002013 001770 001771 001772 001773 002014 002015 004741
000003 001775 001776 002003 002004 002007 001375 001777 001300
000004 002000 001766 002005 002006 001374 000520 001271 001277
000005 001756 001757 001760 001761 001762 001763 120240 120240
000006 120240 120240 120240 120240 120240 120240 120240 120240
000007 120240 120240 120240 120240 120240 000510 010307 010306
000008 010274 010275 010276 102674 002674 000522 000523 000524
000009 000525 000526 001076 000516 000506 000507 000521 000520
000010 000510 000515 000516 001175 001176 001177 001200 000757
000011 001163 000525 000503 000504 000511 000505 000512 010305
000012 001162 001166 001165 000624 000762 001060 000623 000613
000013 000514 000617 000620 000621 000622 001165 010312 000744
000014 001164 005646 001157 000571 010302 010304 010273 010311
000015 000517 000561 000562 000564 010002 000513 001201 007345
000016 107777 007336 007335 007335 007315 007353 006674 000256
000017 000275 006563 177777 120200 000250 000517 000200 005722
000018 005563 005542 000100 000020 000040 000550 140711 154240
000019 140240 147540 151540 141540 123456 123456 123456 123456
000020 123456 123456 123456 123456 123456 123456 123456 123456
*
000400 140265 130264 130260 120240 120240 130663 130262 133260
000401 120240 130663 130266 133263 120240 130663 130262 133266
000402 120240 130663 131662 133263 120240 130662 130262 133266
000403 120240 130663 130266 133262 120240 130663 130262 133262
000404 120240 130663 131662 133262 120240 120240 120240 120240
000405 120240 120240 120240 120240 120240 120240 120240 120240
*
000406 120240 120240 120240 120240 003755 077777 000300 000004
000500 001000 000767 012760 023277 026404 000500 000016 000205
000501 012760 000222 016304 000500 000010 000205 140640 120240
000502 120240 120240 004777 140723 151705 146702 146306 004732
000503 140723 151711 143716 120240 003170 141322 142703 120240
000504 120240 003630 141640 120240 120240 120240 002647 141717
000505 146715 142716 152240 002647 141717 150331 140640 120240
000506 003760 141717 150331 141240 120240 003766 142301 152305
000507 120240 120240 002716 142716 142312 147702 000000 000000
000508 000000 000000 000000 000000 000000 000000 000000 000000
*
000620 000000 000000 000000 000000 000000 000000 120240 120240
000640 003645 143325 147303 152311 147716 003645 144717 146311
000641 151724 120240 003600 145317 141240 120240 120240 002730
000642 142240 120240 120240 120240 004254 146317 140704 120240
000643 120240 004254 151305 153640 120240 120240 003622 151706
000644 144714 142640 120240 003656 151715 140711 147240 120240
000645 005563 151722 142703 120240 120240 003653 151724 140703
000646 145540 120240 002744 152640 120240 120240 120240 004246

```

Note: An asterisk indicates that the succeeding line (or lines) has the same contents as the last printed line.

Figure 2-3. Dump Format

In addition to these task options, the parameter string can also specify relocation values to instruct the system loader where to relocate the program and data. Relocation value parameters are of the form:

$$xx = n$$

where

**xx** is the relocation bias name

**n** is the octal relocation value in the range 0 to 077777 entered as an octal number up to five digits (no leading zero is required).

The relocation bias names are:

**RP** Specifies the program relocation base  
 $0 \leq RP \leq \text{core size}$

**RI** Specifies the indirect pointer relocation base  
 $0 \leq RI \leq 0777$

**RL** Specifies the literal relocation base  
 $0 \leq RL \leq 03777$

**RC** Specifies the COMMON relocation base  
 $0 \leq RC \leq \text{core size}$

If these relocation biases are not specified, /LOAD sets them to the default values defined at system preparation time.

**/ULOAD**  
**/U, name, p(1), p(2), . . . , p(n)**

This control directive directs the executive to call the system loader and load one or more programs from the SF file. Except for the name parameter, the parameters are identical to those of /LOAD.

name specifies the name of the program to be loaded from SF, and is the first parameter in the string. Object module names are generated by DAR MR or FORTRAN IV. Program names are alphanumeric character strings of one to eight characters. Six blanks is an illegal name. Parameters other than name can appear in any order.



## control directives

All error and bounds-checking of parameters is identical to that in /LOAD.

```
/ASSEMBLE  
/A,p(1),p(2),...,p(n)
```

This control directive directs the executive to load the assembler. The parameter string specifies optional tasks for the assembler or executive to perform after the assembly is completed. These tasks are:

| Parameter | Definition                         | Default Assignment                 |
|-----------|------------------------------------|------------------------------------|
| N         | No source listing                  | Source listing                     |
| B         | No binary object<br>program output | Binary object pro-<br>gram listing |
| MAP       | Memory map on load-<br>and-go      | No memory map on<br>load-and-go    |
| L         | Load-and-go after<br>assembly      | No load-and-go after<br>assembly   |
| M         | No symbol table<br>listing         | Symbol table listing               |

If L (load-and-go) is specified, all the options and relocation parameters of /LOAD can be used in /ASSEMBLE. These loading parameters do not apply to the assembly, but to the load-and-go initiated after the assembly is completed.

```
/FORTRAN  
/F,p(1),p(2),...,p(n)
```

This control directive directs the executive to load the FORTRAN compiler. The parameter string specifies optional tasks that the compiler or executive is to perform. These tasks are:

| Parameter | Definition                         | Default Assignment                    |
|-----------|------------------------------------|---------------------------------------|
| N         | No source listing                  | Source listing                        |
| B         | No binary object<br>program output | Binary object pro-<br>gram listing    |
| MAP       | Memory map on load-<br>and-go      | No memory map on<br>load-and-go       |
| L         | Load-and-go after<br>compilation   | No load-and-go after<br>compilation   |
| O         | Octal listing of<br>generated code | No octal listing of<br>generated code |
| X         | Conditional compilation            | No conditional com-<br>pilation       |
| M         | No symbol table listing            | Symbol table listing                  |

| Parameter | Definition                                    | Default Assignment                            |
|-----------|-----------------------------------------------|-----------------------------------------------|
| D         | Generate two-word integer and logical numbers | Generate one-word integer and logical numbers |

**/SMAIN,p(1),p(2)**

This control directive directs the executive to call the system loader and load the system maintenance program.

p(1), if present, is a physical unit name to which PI is assigned. p(2), if present, is a physical unit name to which PO is assigned. Neither the PI nor the PO can be assigned to dummy (DUM).

## DECK PREPARATION

The batch processing facilities of MOS are invoked by control directives in combination with programs and data. These elements form the input job stream to MOS. The input job stream can come from various peripherals and be on various media. The following examples illustrate common job streams and deck preparation.

### Example 1 - Card Input

Request a listing of all logical I/O assignments, enter the current date, set the LO line count to 50, and log a comment to the operator that reads: *MOUNT TAPE DM72*.

```
/JOB,EXAMPLE1
/IOLIST
/DATE,10/15/70
/FORM,50
/C,MOUNT TAPE DM72
/ENDJOB
```

### Example 2 - Card Input

Compile a FORTRAN IV program with source listing, octal object listing, and load-and-go binary output.

```
/JOB,EXAMPLE2
/REW,GO
/FORTRAN,L,O
.
(Source Deck)
.
/EOF
/ENDJOB
```

### Example 3 - Teletype Input

Assign the BI file to magnetic tape unit 2, load a program from BI and produce a map. After loading, halt before execution and produce a dump at program completion.

```
/JOB,EXAMPLE3
/ASSIGN,BI=MT01
/LOAD,MAP,HALT,DUMP
/ENDJOB
```

### Example 4 - Card Input

Copy a card file from PI (processor input, figure 3-1) to scratch 1 (S1). write an end of file, rewind, and copy to the LO file. Then, copy scratch 1 (S1) to the second file of scratch 3 (S3), rewind both scratch tapes, and exit.

```
/JOB,EXAMPLE4
/COPYA,PI=S1
.
(Card File)
.
(End-of-File)
.
/WEOF,S1
/REW,S1
/COPYA,S1=LO
/REW,S1,S3
/SFILE,S3,1
/COPYA,S1=S3
/WEOF,S3
/REW,S1,S3
/ENDJOB
```



## SECTION 3 — INPUT/OUTPUT CONTROL PROGRAM

I/O control is the generalized I/O subsystem under MOS for all users and requires only minimal understanding of the 620 computer hardware I/O operations. All I/O operations, both MOS and user-written, utilize I/O control. During program execution, only the required modules of I/O are loaded into memory.

Because of the standardized interfaces between I/O control and the user's program, and between I/O control and I/O subroutines, I/O peripherals can be changed without program reassembly. As new peripheral devices are added to a system, it is only necessary to program the required I/O driver (appendix A).

Status and error messages are given in section 8.

### LOGICAL AND PHYSICAL UNITS

MOS, through I/O control, allows access to I/O devices and files in terms of logical units with names and/or numbers rather than by actual physical references. The correspondence of a logical unit to a physical unit is made by /ASSIGN prior to program loading and execution.

MOS allows up to 256 logical units, with the first 14 being defined and used by MOS. Figure 3-1 lists logical units defined by MOS; Figure 3-2, physical devices; and Figure 3-3, valid assignments.

| Unit No. | Description     | Unit Name | Function                                                                                                                                                                                                                       |
|----------|-----------------|-----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1        | System file     | SF        | The system file input logical unit. All processing, utility, and library programs are stored here. SF is a magnetic tape unit or drum or disc memory unit.                                                                     |
| 2        | System input    | SI        | The system directive input logical unit. The operating system inputs all of its control directives from SI.                                                                                                                    |
| 3        | System output   | SO        | The system output logical unit. The operating system outputs all input control directives and outputs system operation messages on SO.                                                                                         |
| 4        | Processor input | PI        | The language processor input logical unit. All operating system processors (assembler, compiler, etc.) input source statements from PI.                                                                                        |
| 5        | List output     | LO        | The system listing output logical unit. The operating system outputs all input control directives and any system operations messages on LO. All operating system processors (assembler, compiler, etc.) output listings on LO. |

Figure 3-1. MOS I/O Units (1 of 3)

| Unit No. | Description    | Unit Name | Function                                                                                                                                                                                                                                                                     |
|----------|----------------|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 6        | Binary input   | BI        | The system binary input logical unit. All operating system programs that input binary records input from BI (e.g., loaders).                                                                                                                                                 |
| 7        | Binary output  | BO        | The system binary output logical unit. All operating system processors (assembler, compiler, etc.) that output binary text records output on BO.                                                                                                                             |
| 8        | System scratch | SS        | The system intermediate scratch logical unit. All operating system processors (assemblers, etc.) that use an intermediate scratch unit input from SS.                                                                                                                        |
| 9        | Load and Go    | GO        | The system assemble/compile and GO (load-and-go) logical unit. The assembler and compiler output on GO the same information as output on BO. When assemble/compile and GO is requested, GO references the system resident unit. Otherwise, it references a dummy I/O driver. |
| 10       | Processor      | PO        | The system processor output logical unit. All operating system processors (assembler, etc.) that use an intermediate scratch unit output on PO.                                                                                                                              |

Figure 3-1. MOS I/O Units (2 of 3)



| Unit No. | Description   | Unit Name           | Function                                                        |
|----------|---------------|---------------------|-----------------------------------------------------------------|
| 11       | Scratch 1     | S1                  | System scratch logical unit. For assignment as scratch records. |
| 12       | Scratch 2     | S2                  | System scratch logical unit. For assignment as scratch records. |
| 13       | Scratch 3     | S3                  | System scratch logical unit. For assignment as scratch records. |
| 14       | Scratch 4     | S4                  | System scratch logical unit. For assignment as scratch records. |
| 15-255   | User-assigned | Logical unit number | Can be assigned to any function.                                |

**Figure 3-1. MOS I/O Units (3 of 3)**

| System Name | Physical Device                    |
|-------------|------------------------------------|
| --CPcu      | Card punch                         |
| CRcu        | Card reader                        |
| DKcu        | Disc memory unit                   |
| DRcu        | Drum memory unit                   |
| LPcu        | Line printer                       |
| MTcu        | Magnetic tape unit                 |
| PTcu        | High-speed paper tape reader/punch |
| TPcu        | Teletype paper tape punch          |
| TRcu        | Teletype paper tape reader         |
| TYcu        | Teletype printer                   |
| DUM         | Dummy I/O                          |

**NOTES**

1. cu represents controller/unit.

2.--DUM appears to the I/O control program to be a legitimate device, but in reality does nothing. DUM is used when an I/O output is not desired and for other special system purposes.

Figure 3-2. MOS Physical I/O Devices

| SI | SO | PI  | LO  | BI  | BO  | GO | PO | S1,S2,S3,S4 | SS  |
|----|----|-----|-----|-----|-----|----|----|-------------|-----|
| CR | TY | CR  | CP  | CR  | CP  | DR | DR | ---DR       | CR  |
| DR |    | DR  | DR  | DR  | DR  | MT | MT | ---MT       | DR  |
| MT |    | MT  | LP  | MT  | MT  |    |    | ---DUM      | MT  |
| PT |    | PT  | MT  | PT  | PT  |    |    | ---CR       | PT  |
| TR |    | TR  | PT  | TR  | TP  |    |    | ---PT       | TR  |
| TY |    | TY  | TR  | DUM | DUM |    |    | ---TR       | TY  |
|    |    | DUM | TY  |     |     |    |    | ---TY       | DUM |
|    |    |     | DUM |     |     |    |    | ---LP       |     |

\* For disc versions of MOS, every DRxx unit name is replaced by DKxx.

Figure 3-3. Valid Logical Unit Assignments

| Mnemonic | Definition                | Function Code |
|----------|---------------------------|---------------|
| FUNC     | Perform function          | 016           |
| RALF     | Read alphanumeric record  | 001           |
| RBCD     | Read BCD record           | 041           |
| RBIN     | Read binary record        | 101           |
| REW      | Rewind                    | 004           |
| SKFF     | Skip files forward        | 005           |
| SKFR     | Skip files reverse        | 205           |
| SKRF     | Skip records forward      | 006           |
| SKRR     | Skip records reverse      | 206           |
| STAT     | Request status            | 000           |
| WALF     | Write alphanumeric record | 002           |
| WBCD     | Write BCD record          | 042           |
| WBIN     | Write binary record       | 102           |
| WEOF     | Write end of file         | 003           |

Figure 3-4. Summary of I/O Calls

## I/O CALLS

During program execution, I/O control facilities are accessed through calls specifically defined by the DAS MR assembler. When called, I/O control normally transfers one record at a time between the computer memory and the I/O device. An I/O request to I/O control specifies:

- a. The logical unit number
- b. The type of I/O function to be performed
- c. The number of memory words to be transmitted, or a count for skip operations
- d. The data location

These parameters are combined into 14 I/O calls recognized by the DAS MR assembler

The following abbreviations are used to describe the calls:

| Abbreviation | Definition                                       |
|--------------|--------------------------------------------------|
| DST          | Device specification table                       |
| fc           | Function code                                    |
| IOCS         | I/O control entry point                          |
| loc          | Data address                                     |
| lun          | Logical unit number                              |
| n            | Number of words, records, or files to be skipped |
| wc           | Word count                                       |

### NOTE

The A, B, and X registers and the overflow indicator are assumed volatile and are destroyed during all I/O calls.

input/output control program

The general format of all I/O calls is:

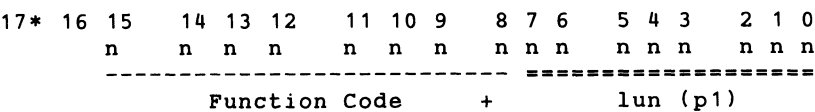
| Label Field       | Operation Field | Variable Field                                                         |
|-------------------|-----------------|------------------------------------------------------------------------|
| Symbol (optional) | Mnemonic        | 1, 2, 3, or 5<br>parameters or ex-<br>pressions separated<br>by commas |

The general format of the expansion of all I/O calls is:

| Label Field       | Operation Field                     | Variable Field                                                     |
|-------------------|-------------------------------------|--------------------------------------------------------------------|
| Symbol (optional) | JMPM<br>DATA<br>.<br>.<br>.<br>DATA | IOCS<br>fc + parameter 1<br>parameter 2<br><br><br><br>parameter n |

where (fc + parameter 1) is:

Bit Position



where n is the octal value of the indicated bit groups.

\* For 18-bit computers (622).

In addition to the expansion of each call, the assembler generates two other directives at the beginning of every program that uses I/O calls:

```
EXT      IOCS
ION      lun 1,...,lun n
```

The EXT declares IOCS an external reference, while ION does the same for the necessary drivers. Both allow the loader to link the user and I/O control at load time.

The following sections give a detailed discussion of each I/O call defined within MOS.

**READ BINARY RECORD**

Call:

|             |                   |
|-------------|-------------------|
| <b>RBIN</b> | <b>lun,wc,loc</b> |
|-------------|-------------------|

Expansion:

|             |                   |
|-------------|-------------------|
| <b>JMPM</b> | <b>IOCS</b>       |
| <b>DATA</b> | <b>040400+lun</b> |
| <b>DATA</b> | <b>wc</b>         |
| <b>DATA</b> | <b>loc</b>        |

This call inputs wc words from the I/O device to consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If the input record contains more than wc words, only wc words are stored in memory, and the remainder, ignored. If the input record contains less than wc words, they are input. The number of words is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

**READ ALPHANUMERIC RECORD**

Call:

|             |                   |
|-------------|-------------------|
| <b>RALF</b> | <b>lun,wc,loc</b> |
|-------------|-------------------|

Expansion:

|             |                   |
|-------------|-------------------|
| <b>JMPM</b> | <b>IOCS</b>       |
| <b>DATA</b> | <b>000400+lun</b> |
| <b>DATA</b> | <b>wc</b>         |
| <b>DATA</b> | <b>loc</b>        |

This call inputs wc words from the I/O device to consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If the input record contains more than wc words, only wc words are stored in memory, and the remainder, ignored. If the input record contains less than wc words, they are input. The number of words is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

**READ BCD RECORD**

Call:

|             |                   |
|-------------|-------------------|
| <b>RBCD</b> | <b>lun,wc,loc</b> |
|-------------|-------------------|

Expansion:

|             |                   |
|-------------|-------------------|
| <b>JMPM</b> | <b>IOCS</b>       |
| <b>DATA</b> | <b>020400+lun</b> |

## input/output control program

|      |     |
|------|-----|
| DATA | wc  |
| DATA | loc |

This function inputs wc words from the I/O device to consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If the input record contains more than wc words, only wc words are stored in memory, and the remainder, ignored. If the input record contains less than wc words, they are input. The number of words input is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

## WRITE BINARY RECORD

Call:

|      |            |
|------|------------|
| WBIN | lun,wc,loc |
|------|------------|

Expansion:

|      |            |
|------|------------|
| JMPM | IOCS       |
| DATA | 041000+lun |
| DATA | wc         |
| DATA | loc        |

This function outputs wc words to the I/O device from consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If output is specified with less than wc words, they are output. If the configuration permits wc or more words, wc words are output. The number of words output is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

## WRITE ALPHANUMERIC RECORD

Call:

|      |            |
|------|------------|
| WALF | lun,wc,loc |
|------|------------|

Expansion:

|      |            |
|------|------------|
| JMPM | IOCS       |
| DATA | 001000+lun |
| DATA | wc         |
| DATA | loc        |

This function outputs wc words to the I/O device from consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If output is specified with less than wc words, they are output. If the configuration permits wc or more words, wc words are output. The number

of words output is placed in word 0 of the DST. The I/O request is ignored if wc is not greater than zero.

## WRITE BCD RECORD

Call:

```
WBCD      lun,wc,loc
```

Expansion:

```
JMPM      IOCS
DATA      021000+lun
DATA      wc
DATA      loc
```

This function outputs wc words to the I/O device from consecutive memory addresses beginning at loc. The function is performed in the mode requested, if possible. Otherwise, the mode of the device is used. If output is specified with less than wc words, they are output. If the configuration permits wc or more words, wc words are output. The number of words output is placed in word 0 of the DST. The request is ignored if wc is not greater than zero.

### Note:

The high-speed paper-tape driver for reading and writing unformatted paper tapes makes no distinction between RBIN and RALF when reading, or between WBIN and WALF when writing. The driver will read one record which is defined as the number of words requested by the user program, or 60 words (120 frames) whichever is smaller. Each record is checked for leading blank characters. If a length of tape equal to one-half record is read from the beginning of the record as contiguous blank characters, the blank area is interpreted as an end-of-file. The driver will write two frames per word on paper tape for as many words as are specified by the user program. The DST and paper tape driver are combined into one routine labeled \$0P which corresponds to the peripheral mnemonic PT10.

## WRITE END OF FILE

Call:

```
WEOF      lun
```

Expansion:

```
JMPM      IOCS
DATA      001400+lun
```



## input/output control program

This function outputs an end-of-file character to the peripheral device. No data are transmitted. Subsequent I/O status requests produce an end-of-file return.

### REWIND

Call:

REW            1un

Expansion:

JMPM           IOCS  
DATA           002000+1un

This function issues a rewind command to the I/O device. No data are transmitted. Subsequent I/O status requests produce a beginning-of-device return.

### SKIP RECORDS FORWARD

Call:

SKRF           1un,n

Expansion:

JMPM           IOCS  
DATA           003000+1un  
DATA           n

This function skips n records in the forward direction on the I/O device. No data are transmitted. If an end of device or end of file is detected before the requested number of records are skipped, skipping terminates and the count of records remaining to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce an end-of-device or end-of-file return, respectively. The request is ignored if n is not greater than zero.

### SKIP RECORDS REVERSE

Call:

SKRR           1un,n

Expansion:

JMPM           IOCS  
DATA           010300+1un  
DATA           n

This function skips  $n$  records in the reverse direction on the I/O device. No data are transmitted. If a beginning of device or an end of file is detected before the requested number of records are skipped, skipping terminates and the count of records remaining to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce a beginning-of-device or end-of-file return, respectively. The request is ignored if  $n$  is not greater than zero.

## SKIP FILES FORWARD

Call:

**SKFF**                     $1un, n$

Expansion:

**JMPM**                    IOCS  
**DATA**                     $002400+1un$   
**DATA**                     $n$

This function skips  $n$  file marks in the forward direction on the I/O device. No data are transmitted. Subsequent I/O status requests produce an end-of-file return. If an end of device is detected before the requested number of files are skipped, skipping terminates and the count of any remaining files to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce an end-of-device return. The request is ignored if  $n$  is not greater than zero.

## SKIP FILES REVERSE

Call:

**SKFR**                     $1un, n$

Expansion:

**JMPM**                    IOCS  
**DATA**                     $010240+1un$   
**DATA**                     $n$

This function skips  $n$  file marks in the reverse direction on the I/O device. No data are transmitted. Subsequent I/O status requests produce an end-of-file return. If an end of device is detected before the requested number of files are skipped, skipping terminates and the count of any remaining files to be skipped is placed in word 0 of the DST. Subsequent I/O status requests produce a beginning-of-device return. The request is ignored if  $n$  is not greater than zero.

**PERFORM FUNCTION**

Call:

|             |               |
|-------------|---------------|
| <b>FUNC</b> | <b>1un, n</b> |
|-------------|---------------|

Expansion:

|             |                   |
|-------------|-------------------|
| <b>JMPM</b> | <b>IOCS</b>       |
| <b>DATA</b> | <b>003400+1un</b> |
| <b>DATA</b> | <b>n</b>          |

This function commands one of the following special functions peculiar to the specified I/O device:

| Peripheral                         | Function                                                                                                                                                                        |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Teletype keyboard (TYcu)           | Spaces paper five lines (n ignored)                                                                                                                                             |
| Teletype paper tape punch (TPcu)   | Punches about 24 inches of blank leader (n ignored)                                                                                                                             |
| High-speed paper tape punch (PTcu) | Punches about 24 inches of blank leader (n ignored)                                                                                                                             |
| Card punch (CPcu)                  | Ejects one blank card (n ignored)                                                                                                                                               |
| Line printer (LPcu)                | Slews paper at a rate other than one line per print line (n is the slew character). The page position for each of the eight possible counts in a function call is listed below: |

| Value of n | Channel on Format Tape | Position on Page  |
|------------|------------------------|-------------------|
| 0          | 0                      | Top of form       |
| 1          | 1                      | Reserved for user |
| 2          | 2                      | Reserved for user |
| 3          | 3                      | Reserved for user |
| 4          | 4                      | Reserved for user |
| 5          | 5                      | Reserved for user |
| 6          | 6                      | Reserved for user |
| 7          | 7                      | Reserved for user |

## REQUEST STATUS

Call:

**STAT**                    **lun, err, eof, beod, busy**

Expansion:

|             |              |
|-------------|--------------|
| <b>JMPM</b> | <b>IOCS</b>  |
| <b>DATA</b> | <b>0+lun</b> |
| <b>DATA</b> | <b>ERR</b>   |
| <b>DATA</b> | <b>EOF</b>   |
| <b>DATA</b> | <b>BEOD</b>  |
| <b>DATA</b> | <b>BUSY</b>  |

This function examines the I/O driver of a logical unit to determine its status and then specifies return addresses to be used depending on this status. Return addresses can specify indirect addressing. On return, the X register points to word 0 of the DST of the I/O driver for this lun. If this lun has no I/O driver, the X register is cleared.

The various exit terms are:

|             |                                          |
|-------------|------------------------------------------|
| <b>ERR</b>  | I/O error on last transfer               |
| <b>EOF</b>  | End of file on last transfer             |
| <b>BEOD</b> | Beginning/end of device on last transfer |
| <b>BUSY</b> | Device busy                              |

## I/O PROGRAMMING EXAMPLES

### Example 1

Rewind tape unit 3 (lun = 15) and read one file consisting of 100 records of 60 binary words each. After each record is read, print it on the line printer (lun = 5). Go to the top of the next form upon completion and rewind tape unit 3. Halt on an end of file or I/O error. Exit to the resident monitor on normal completion.

| Label Field | Operation Field | Variable Field |
|-------------|-----------------|----------------|
| A           | REW             | 15             |
|             | LDAI            | -100           |
| B           | RALF            | 15,60,BUF      |
| C           | STAT            | 15,X,X,X,C     |
|             | WALF            | 5,60,BUF       |
| D           | STAT            | 5,X,X,X,D      |
|             | IAR             |                |
|             | JAN             | B              |
|             | FUNC            | 5,1            |
| E           | STAT            | 5,X,X,X,E      |
|             | REW             | 15             |
| F           | STAT            | 15,X,X,G,F     |
| BUF         | BSS             | 60             |
| X           | HLT             | 0              |
|             | EXT             | EXIT           |
| G           | CALL            | EXIT           |
|             | END             | A              |

### Example 2

Read a card from the card reader (lun = 6) until an end of file is detected and write it on a drum file (lun = 20). List all I/O errors on the Teletype, but do not terminate the operation.

| Label Field | Operation Field | Variable Field |
|-------------|-----------------|----------------|
| A           | RBIN            | 6,60,BUF       |
| B           | STAT            | 6,F,D,F,B      |
|             | WBIN            | 20,60,BUF      |
| E           | STAT            | 20,F,F,F,E     |
|             | JMP             | A              |
| F           | WALF            | 3,3,H          |
| J           | STAT            | 3,A,A,A,J      |
|             | JMP             | A              |
|             | EXT             | EXIT           |
| D           | CALL            | EXIT           |
| BUF         | BSS             | 60             |
| H           | DATA            | 'IO ERR'       |
|             | END             | A              |

### Example 3

Read a disc file (lun = 17) consisting of 40-word records until and end of file is detected. Search each record for a zero word and keep a count of them. At end of file, punch a binary card on lun = 7 with the count of zero words in the first card word. Ignore I/O errors.

| Label Field | Operation Field | Variable Field |
|-------------|-----------------|----------------|
| A           | TZA             |                |
|             | STA             | H              |
| B           | RBIN            | 17,40,BUF      |
| D           | STAT            | 17,C,G,C,D     |
| C           | LDXI            | BUF            |
| E           | LDA             | 0,X            |
|             | XAZ             | F              |
|             | INCR            | 045            |
|             | SUBI            | BUF + 40       |
|             | JAP             | B              |
|             | JMP             | E              |
| G           | WBIN            | 7,1,H          |
| I           | STAT            | 7,J,J,J,I      |
|             | EXT             | EXIT           |
| J           | CALL            | EXIT           |
| F           | INR             | H              |
| H           | DATA            | 0              |
| BUF         | BSS             | 40             |
|             | END             | A              |



## SECTION 4 – DEBUGGING PROGRAM

The MOS debugging program aids the programmer in finding and correcting program errors. Its commands examine and/or change the program, in addition to running part or all of a program.

Whenever the DEBUG option is specified on /LOAD or /ULOAD (section 2), the debugging program is loaded with the user's program. When loading is complete, control is transferred to the debugging program.

Status and error messages are given in section 13.

### DIALOG

The debugging program is an interactive component within MOS used via SI. Upon entry, if SI = TY00, it types:

**C/R\*\***

To communicate with the debugging program, use the command language with the following syntax:

- a. General form:  
**instruction parameter(1),parameter(2),...,parameter(n)**
- b. Instructions can have up to 72 characters. Characters beyond the 72nd are ignored.
- c. Continuation lines begin with a comma (,).
- d. Invalid instructions cause the reply **WHAT** followed by \*\* if SI = TY00.
- e. All numbers are octal.
- f. Negative (two's complement) numbers are preceded by a minus (-) sign.



## EXAMPLES OF DEBUGGING

In the following examples, symbols in bold type indicate user-entered information. Other symbols represent the output of the debugging program.

### Display and Alter Instructions

```

**A                      Display the contents of the pseudo-A register.
(077553)                Contents of the pseudo-A register.
**A005572              Change the contents of A to 005572.
**B                    Display the contents of the pseudo-B register.
(177750)                Contents of the pseudo-B register.
**B-1                  Change the contents of B to minus one (0177777)
                        (0777777 on 622 computers)
**X                    Display the contents of the pseudo-X register.
(000021)                Contents of the pseudo-X register.
**X12                  Change the contents of X to 000012.
**O                    Display the contents of the pseudo-overflow
                        indicator.
(000000)                Contents of the pseudo-overflow indicator.
**O-1                  Set the pseudo-overflow indicator.
**26001                Display the contents of memory address 026001.
(170522)                Contents of memory address 026001.
**1002,1045            Display the contents of memory addresses 001002
                        through 001045.

```

|        |        |        |        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 001000 | 177776 | 100001 | 010101 | 151501 | 025252 | 000000 | 000101 | 015432 |
| 001010 | 144456 | 052345 | 177777 | 101010 | 111101 | 063063 | 033333 | 177777 |
|        | .      |        | .      |        | .      |        | .      |        |
|        | .      |        | .      |        | .      |        | .      |        |
|        | .      |        | .      |        | .      |        | .      |        |
| 001040 | 177776 | 177776 | 177775 | 020205 | 123456 | 000000 | 035353 | 077756 |

```

**C26000,1000,2500      Change the contents of addresses 026000 through
** ,77777,375           026003 to 001000, 002500, 077777, and
                        000375, respectively.
**|1000,1100,125252     Initialize addresses 001000 through 001100 to
                        0125252 (each word from 001000 through

```

001100 contains 0125252).

**\*\*S5000,6000,100000**

Search addresses 005000 through 006000 for 0100000 and print each address where this value is found.

005100  
005302  
005701

In this example, 0100000 was found in addresses 005100, 005302, and 005701

**\*\*S6300,10000,136000,177000**

Search addresses 006300 through 010000 for the value 0136 in bits 9 through 15 of each word and print each such address and its full contents. The last parameter, 017700, is used as a mask.

006320.  
006700  
006701

In this example, three locations met the comparison criteria.

(136111)  
(136111)  
(136000)

### I/O Instructions

**\*\*ASSIGN,12 = MT01**

Assign logical unit 12 to device MT01 (magnetic tape unit 01). The unit number is decimal.

**\*\*IOLIST**

List the assignment of logical units to physical devices.

01 Blank  
02 CR00  
03 TY00  
04 Blank  
.  
.  
.  
.  
15 MT03

In this example, LUN 2 is the card reader, LUN 3, is the Teletype, and LUN 15 is magnetic tape unit 03. LUN 1 and LUN 4 were not loaded with DEBUG.

**\*\*W17000,17556,17000,TEST**

Write the program named TEST on the BO in absolute format (TEST resides in addresses 017000 through 017556) and set execution starting at 017000.

**\*\*W14000,14100,TEST**  
**\*\*W14700,15334**

Write the program named TEST on the BO in absolute format (TEST resides in addresses

| PAGE | 1     | RSCR2LPC | 11/06/70 |     |     |     |     |     |     |
|------|-------|----------|----------|-----|-----|-----|-----|-----|-----|
| A    | SRIT  | 52       | AK       | 121 | 123 | 125 | 127 | 135 | 137 |
| 0    | SJCW  | 17       | 1A       |     |     |     |     |     |     |
| 0    | SLAX  | 18A      | 180      |     |     |     |     |     |     |
| 0    | SLRF  | 157      | 15A      | 159 |     |     |     |     |     |
| 0    | SROC  | 11A      | 117      |     |     |     |     |     |     |
| 193  | SXEQ  | 192      |          |     |     |     |     |     |     |
| 159  | ASRF  | 43       | 4A       | 6A  | 73  | 96  | 100 |     |     |
| 10   | HASF  | 15       |          |     |     |     |     |     |     |
| 191  | RLK   | 40       | 143      |     |     |     |     |     |     |
| 15A  | BLFA  | 30       | 49       | 72  | 74  | 101 | 151 | 154 |     |
| 92   | CHAR  | 87       |          |     |     |     |     |     |     |
| 104  | D100  | 17A      |          |     |     |     |     |     |     |
| 70   | D60   | 44       |          |     |     |     |     |     |     |
| 24   | DMP   | 1A       | 14       |     |     |     |     |     |     |
| 3A   | DMPG  | 112      |          |     |     |     |     |     |     |
| 143  | DPA   | 146      |          |     |     |     |     |     |     |
| 27   | DSCW  | 142      |          |     |     |     |     |     |     |
| 114  | DSDC  | 109      |          |     |     |     |     |     |     |
| 185  | DUDE  | 11A      |          |     |     |     |     |     |     |
| 122  | DSDL  | 26       | 136      | 138 | 14A |     |     |     |     |
| 50   | DSFB  | 71       | 99       |     |     |     |     |     |     |
| 140  | DWHK  | 134      |          |     |     |     |     |     |     |
| 149  | DWIT  | 111      | 131      | 144 | 155 |     |     |     |     |
| 14A  | JMP   | 124      |          |     |     |     |     |     |     |
| A    | LPT   | 47       | 161      |     |     |     |     |     |     |
| 18A  | KDDMP | 12       | 21       |     |     |     |     |     |     |
| 160  | C10   | 20       |          |     |     |     |     |     |     |
| 173  | D200  |          |          |     |     |     |     |     |     |
| 163  | PAGE  | 2A       | 165      | 166 | 176 | 179 | 185 |     |     |
| 1A   | RSCR2 | 194      |          |     |     |     |     |     |     |
| 16A  | SEMS  | 70       | 82       | 8A  | 93  | 102 | 164 | 169 | 170 |
|      |       | 182      | 183      |     |     |     |     |     | 175 |
| 15A  | TEMP  | 3E       | 7A       | 107 | 115 | 150 | 153 |     | 177 |
| 172  | U111  | 162      |          |     |     |     |     |     |     |
| 184  | VVV   | 171      |          |     |     |     |     |     |     |
| 10A  | WWW   | 80       |          |     |     |     |     |     |     |
| 81   | XXX   | 46       |          |     |     |     |     |     |     |
| 63   | YYY   | 58       |          |     |     |     |     |     |     |
| 50   | ZZZ   | 53       |          |     |     |     |     |     |     |

Figure 5-1. Sample Concordance Listing

Beginning with the first character position, the format for a concordance line is:

- a. Four positions to display the decimal line number where the symbol is defined. The line number is right-justified and left-blank-filled.
- b. Two blanks.
- c. Six positions to display the symbol, in sort sequence, left-justified and right-blank-filled.
- d. Up to ten reference line numbers. Each reference line number is in sort sequence, preceded by two blanks and four character positions to display the decimal line number, right-justified and left-blank-filled.

The maximum line is 72 characters. Continuation lines contain only the reference line numbers.

Status and error messages are given in Section 13.



## SECTION 6 — FILE EDITING PROGRAM

The file editing program is a support program operating under MOS that is a simple and powerful tool for editing files generated under MOS (e.g., source programs, text, etc.). The user supplies the file editing program with control directives and files, and the program produces new and/or modified files and an audit trail of all transactions.

Status and error messages are given in Section 13.

The file editing program accepts single-reel files and multifile reels of input. It processes input in control directive and source file format, and outputs source file format and a printed audit listing.

Control directives are input through the SI.

On being loaded by /ULOAD,EDITOR,RP = 0500,RI = 0377 (Section 2), the file editing program types

### BEGIN EDITOR

on the LO and inputs control directives to obtain its instructions from the user. The following conventions are used in describing and coding file editing directives:

- a. General form:

$n, p(1), p(2), \dots, p(x)$

where  $n$  is the directive name, and  $p(1), p(2), \dots, p(x)$  is a parameter string with individual parameters separated by commas.

- b. Directives begin in the first character position of the record and can consist of up to 72 characters.
- c. All fields are interpreted as fixed-format data and must be the exact size as indicated below.
- d. Parameters shown in upper-case letters appear on the control records exactly as shown.

- e. Parameters in italics are optional.
- f. Parameters in lower-case letters are to be replaced by user-defined character strings according to the following:

|          |                                                                                                                                 |
|----------|---------------------------------------------------------------------------------------------------------------------------------|
| filename | Name of file: any <b>eight</b> characters                                                                                       |
| author   | Author of file: any <b>12</b> characters                                                                                        |
| aaa      | Alphabetic portion of sequence number: any <b>three</b> alphabetic characters                                                   |
| nnnnn    | Numeric portion of sequence number: any <b>five</b> numeric digits ending in zero                                               |
| location | Either the internal sequence number (in the form aaannnnn) or the external line number (in the form nnnnn) identifying a record |
| nnn      | Number of records: any <b>three</b> numeric digits                                                                              |
| l        | The letter l at the end of a parameter string indicates that location is an internal sequence number.                           |

## **DIRECTIVES**

### **LIST,xxx**

LIST specifies list output during editing. If omitted, no list output is provided. Its presence, with or without a parameter, causes a file audit and outputs catalog to be printed (figure 6-1). The parameter xxx, if used, is:

- a. *CHG* - list changes only
- b. *ALL* - list complete file(s)

LIST can be repeated for different options on different files. It immediately precedes a FILE, COPY, or EDIT.

When an audit listing is output, it is produced on the LO. Three levels of reporting are provided. All include a heading identifying the report as an output and giving the run date (Figure 6-2).

|               |                |          |          |
|---------------|----------------|----------|----------|
| FILE EDIT RUN | OUTPUT CATALOG | 10/14/70 | PAGE0003 |
| FILE NAME     | # OF RECORDS   | CHANGED  |          |
| NUM1          | 0008           | 0287     |          |
| NUM2          | 0013           | 0287     |          |

\*\*

**Figure 6-1. Output Catalog Format**



|                  |                 |                           |          |
|------------------|-----------------|---------------------------|----------|
| FILE EDIT RUN    | FILE AUDIT      | 10/14/70                  | PAGE0002 |
| FILENAME IS NUM2 | AUTHOR IS JONES | CREATION DATE IS 10/14/70 |          |

| *CONTROL RECORD OR DATA RECORD IMAGE* |      |                         | LINE#    | DATE  | CODE     |
|---------------------------------------|------|-------------------------|----------|-------|----------|
| C                                     | WBIN | BO, 10, D               | VDM00000 | 00001 | 0287 ADD |
|                                       | STAT | BO, ERR, EOF, BEOD, *-6 | VDM00010 | 00002 | 0287 ADD |
|                                       | EXT  | E                       | VDM00020 | 00003 | 0287 ADD |
|                                       | JMP  | E                       | VDM00030 | 00004 | 0287 ADD |
| D                                     | BSS  | 10                      | VDM00040 | 00005 | 0287 ADD |
| BO                                    | EQU  | 7                       | VDM00050 | 00006 | 0287 ADD |
| ERR                                   | INCR | 01                      | VDM00060 | 00007 | 0287 ADD |
| EOF                                   | INCR | 02                      | VDM00070 | 00008 | 0287 ADD |
| BEOD                                  | INCR | 04                      | VDM00080 | 00009 | 0287 ADD |
|                                       | EXT  | F                       | VDM00090 | 00010 | 0287 ADD |
|                                       | JMP  | F                       | VDM00100 | 00011 | 0287 ADD |
|                                       | END  | C                       | VDM00110 | 00012 | 0287 ADD |
| /ENDJOB                               |      |                         | VDM00120 | 00013 | 0287 ADD |

## NOTES

1. CONTROL RECORDS ARE PRECEDED AND FOLLOWED BY TWO BLANK LINES.
2. GROUPS OF ADDITIONS OR DELETIONS ARE PRECEDED AND FOLLOWED BY ONE BLANK LINE.
3. EACH NEW FILE BEGINS ON A NEW PAGE.
4. LITING OUTPUT SETUP USING TELETYPE AS LO.

Figure 6-2. Audit Listing Format

The output catalog report is a summary of the contents of the source library showing the file names and the number of records each contains. The date of the latest change to each file (excluding resequencing only) is shown.

The file audit report is available in two levels for each file on the source library. All records in the file can be shown, or the report can be limited to those records having changes. Each record listed is accompanied by a program-generated line number and an action code (ADD) if the record has been added. All control records are also listed, and have four asterisks in the action code field.

**FILE,filename,author,SEQ,aaannnnn**

FILE creates a file from the data records following it. The file is formatted as a standard MOS source file (section 3). The optional SEQ parameter assigns internal sequence numbers in positions 73 through 80 of each record, beginning with the value specified in aaannnnn and incrementing by ten for each record.

FILE is followed immediately by the data records comprising the file.

An end of file following the data records indicates the end of the data to be processed with FILE. This can be a 2-7-8-9 punch on cards; a BELL character on the Teletype; or an EOF mark on paper tape, magnetic tape, drum, or disc.

**COPY,filename,filename,SEQ,aaannnnn**

COPY copies a file or files from the input device to the output device. If a second filename is specified, all files from the one specified in the first filename through that of the second filename are copied. If SEQ and aaannnnn are present, they apply only to the first file being copied and cause the program to assign internal sequence numbers in positions 73 through 80 of each record being copied. Sequence numbers start with aaannnnn and increment by ten for each record.

**EDIT,filename,SEQ,aaannnnn**

EDIT copies a file from the input device to the output device with the modifications specified by the ADD, DEL, and RPL that immediately follow EDIT. If SEQ and aaannnnn are present, they apply only to the first file being copied and cause the program to assign internal sequence number in positions 73 through 80 of each record being copied. Sequence numbers start with aaannnnn and increment by ten for each record.

**ADD,nnn,location,l**

ADD adds the following nnn data records to the file being processed immediately after the record specified by location. If location represents an internal sequence number rather than the line number of the previous audit list, add l.

This control record is valid only after an EDIT and before an ENDCOR.

### **DEL,location,location,l**

DEL deletes a record or records from the file being processed. If a second location is present, all records from the one specified in the first location through that of the second are deleted. If location represents an internal sequence number rather than the line number of the previous audit list, add l.

This control record is valid only after an EDIT and before an ENDCOR.

### **RPL,nnn,location,location,l**

RPL deletes a record or records from the file being processed and replaces them with the following nnn data records. If a second location is present, all records from the one specified in the first through that of the second are deleted and replaced with the following nnn data records. If location represents an internal sequence number rather than the line number of the previous audit list, add l.

This control record is valid only after an EDIT and before an ENDCOR.

### **ENDCOR**

ENDCOR indicates the end of modifications to the file in process. It copies the remainder of the file to be copied to the output device without modification.

## **SOURCE FILE**

When a source file is to be copied or edited, it is input through the PI. The new or modified file is output by the PO.

A file in a source-language library consists of a header record, an EOF, the data records comprising the file, another EOF, and a catalog of all preceding files including the current one.

If a file follows another file, its header record immediately follows the catalog at the end of the preceding file. The last file on the source library is followed by two EOF records. Figure 6-3 shows the structure of a typical MOS source file.

### **HEADER RECORD**

The header record is the first record of each file. It is generated with the file by FILE and is used as the identifier for the file. It is 14 words (28 characters) long in the following format:

|          |          |           |
|----------|----------|-----------|
| 1-----8  | 9-----20 | 21-----28 |
| FILENAME | AUTHOR   | DATE      |

where the filename and author are taken from FILE parameters and date is the creation date as input on the /DATE card to the executive (Section 2).

DATA RECORD

A data record is a fixed-length record of 46 words in the following format:

1-----80 81-----88      89-----92  
CHARACTER DATA      BLANK      yddd

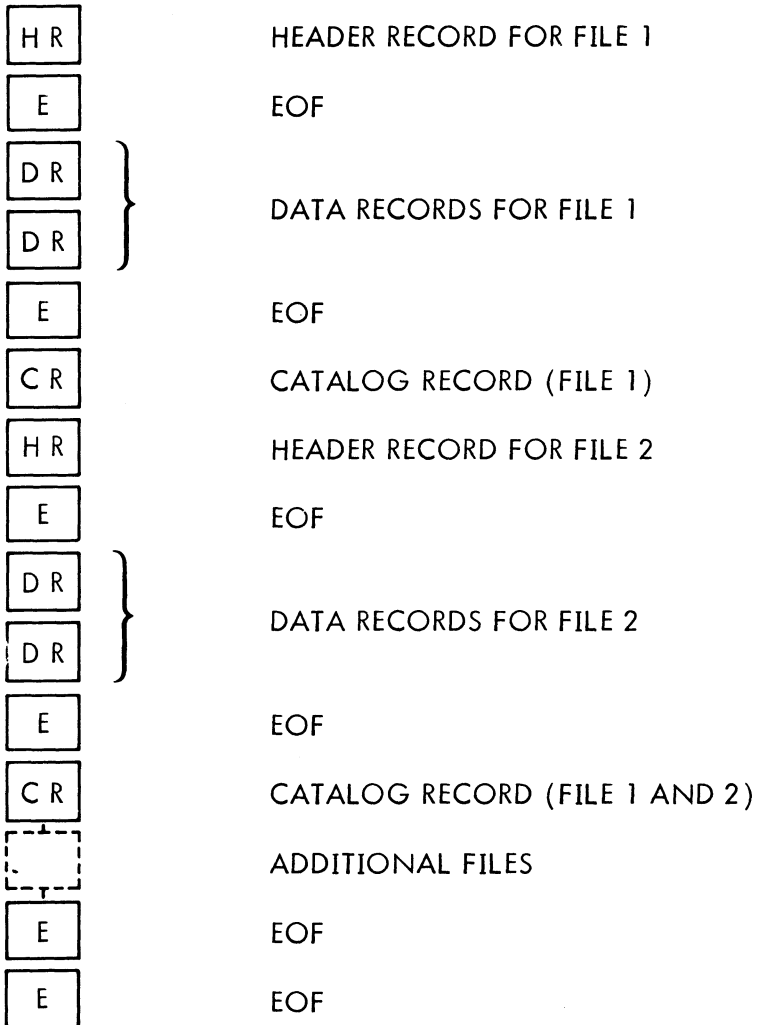
where yddd is the date (units position of year followed by the day) of the last change to the record (excluding resequencing).

CATALOG RECORD

The catalog record is a fixed-length record of eight words per entry, and up to 20 entries. Each catalog record contains an entry for each preceding file. Catalog records have the following format:

FILENAME  
COUNT                      File 1  
yddd  
FILENAME  
COUNT                      File 2  
yddd  
                             .  
                             .  
                             .  
                             File n

where the filename is taken from FILE, count is a two-word binary integer representing the number of records in the file, and yddd is the date (units position of the year followed by the day) of the last change to the record (excluding resequencing).

**Figure 6-3. MOS Source File Structure**

## SECTION 7 – SYSTEM MAINTENANCE PROGRAM

The system maintenance program (SMP) is an MOS support program for use in updating an MOS installation system library (ISL) prior to the use of the ISL in system preparation. As input, the user supplies the SMP with control directives and an ISL. The program outputs a new library and optional directive listings of both old and new libraries.

Status and error messages are given in Section 13.

The SMP is loaded with /SMAIN (Section 2), which also specifies the logical input and output units. On loading, the program types the message:

### BEGIN SYSTEM MAINTENANCE

on the Teletype (and LO, if different). All control directives are then input through SI before processing of the ISL is begun.

SMP directives have the following specifications:

- a. General form:

name,p(1),p(2),...,p(n)

where name is the directive name and p(1),p(2),...,p(n) is a parameter string with individual parameters separated by commas.

- b. Directives have up to 72 characters and begin in the first character position of the record (card).
- c. Embedded blanks are allowed within a field.
- d. Fields smaller than the maximum are left-justified and blank-filled.

### DIRECTIVES

**ADD,name(1),name(2),...,name(n)**

## **system maintenance program**

ADD adds a program or group of programs to the new ISL. The programs contained on the BI logical unit are added selectively to the new ISL after each program specified by ADD. Each name is the name of a program in the old ISL.

In processing ADD, the SMP copies from the old ISL to the new ISL until it has copied the program specified by a name in ADD. Then the SMP types the message:

**ADD name**

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

- Y** Add the program contained on the BI, copy it into the new ISL, and return for another control directive.
- N** The additions are complete. Resume copying to the next name. N also types the message:

**END ADDITIONS**

**DELETE,name(1),name(2),...,name(n)**

DELETE deletes the specified program or programs from the old ISL when the new ISL is produced. Each name is the name of a program in the old ISL.

In processing DELETE, the SMP copies from the old ISL to the new ISL all programs except those named in DELETE.

**REPLACE,name(1),name(2),...,name(n)**

REPLACE replaces a program or programs from the old ISL. The program(s) specified in REPLACE are deleted and selectively replaced by the new program(s) contained on the BI. Each name is the name of a program in the old ISL.

In processing REPLACE, the SMP copies from the old ISL to the new ISL until it has copied the program preceding the program specified by a name in REPLACE. The SMP skips the specified program(s) on the old ISL and types the message:

**REPLACE name**

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

- Y** Add the program contained on the BI, copy it into the new ISL, and return for another control directive.

- N** The replacements are complete. Resume copying to the next name. N also types the message:

### **END REPLACEMENTS**

#### **END,p**

END initiates the ISL copying process. All necessary ADD, DELETE, and REPLACE directives are input before END. END rewinds the old and new ISL media, starts the copying process, and continues until it reaches the end of the old ISL.

END control directive parameter p is either:

- |          |                                                       |
|----------|-------------------------------------------------------|
| <b>L</b> | List the old ISL on the LO                            |
| blank    | Omit the listing (in this case, no comma follows END) |

#### **LIST**

**L** or **LIST** lists on LO the ISL on the PI (Figure 7-1). A single letter (A or R), to the left of a program name under the ID header, identifies the program as an absolute or relocatable module.

#### **DATE,xxxxxxx**

DATE sets up to eight alphanumeric characters (e.g., a date) in the page title of the list output. If DATE is not used, the date is used from the last /DATE input to the executive



PAGE 1 01/19/71

BEGIN SYSTEM MAINTENANCE  
END,L

PAGE 2 01/19/71

| ID                                | DATE     | SIZE  | ENTRY NAMES                                                                                                                                                             | EXTERNAL NAMES                                     |
|-----------------------------------|----------|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------|
| A PREP1                           | 01/14/71 | 17435 |                                                                                                                                                                         |                                                    |
| A PREP2                           | 01/12/71 | 25377 |                                                                                                                                                                         |                                                    |
| ABS,RSCB1DK,05/12/70,-1,01000,010 |          |       |                                                                                                                                                                         |                                                    |
| R RSCB1RMC                        | 01/12/71 | 00721 | CYLN04 CYLN01<br>CYLN00 \$XEQ<br>\$UCB4 \$UCB1<br>\$UCB0 \$TTL<br>\$ROC \$REW<br>\$PED \$PCW<br>\$MPR \$LCB<br>\$LBF \$LAX<br>\$LAB \$JCH<br>\$FRM \$ECW<br>\$DAT \$RUF |                                                    |
| ENDABS                            |          |       |                                                                                                                                                                         |                                                    |
| ABS,RSCB1MT,01/11/70,-1,01000,010 |          |       |                                                                                                                                                                         |                                                    |
| R RSCB1MTC                        | 00662    |       | CYLN04 CYLN01<br>CYLN00 \$XEQ<br>\$UCB4 \$UCB1<br>\$UCB0 \$TTL<br>\$ROC \$REW<br>\$PED \$PCW<br>\$MPR \$LCB<br>\$LBF \$LAX<br>\$LAB \$JCH<br>\$FRM \$ECW<br>\$DAT \$RUF |                                                    |
| ENDABS                            |          |       |                                                                                                                                                                         |                                                    |
| ABS,RSCB2TY,01/11/70,-1,01000,010 |          |       |                                                                                                                                                                         |                                                    |
| R RSCB2TY                         | 03/19/70 | 00166 | \$XEQ                                                                                                                                                                   | \$RNC \$LAX \$JCH \$FRM                            |
| ENDABS                            |          |       |                                                                                                                                                                         |                                                    |
| ABS,RSCB2LP,01/11/70,-1,01000,010 |          |       |                                                                                                                                                                         |                                                    |
| R RSCB2LP                         | 10/29/70 | 00215 | \$XEQ                                                                                                                                                                   | \$RNC \$LBF \$LAX \$JCH                            |
| ENDABS                            |          |       |                                                                                                                                                                         |                                                    |
| ABS,RSCB3,01/11/70,-1,01000,010   |          |       |                                                                                                                                                                         |                                                    |
| R RSCB3                           | 00435    |       | IOCS EXIT<br>EXIT \$RT<br>\$PUN \$PGM<br>\$MOS \$LIN<br>\$LIT \$IOW                                                                                                     | \$XEQ \$REW \$PUB \$LUB<br>\$LAX \$JCH \$ECW \$RUF |

Figure 7-1. System Maintenance List Output

## **SECTION 8 – SYSTEM PREPARATION PROGRAM**

The system preparation program (SPP) is a stand-alone support program of MOS. It creates a system file on a magnetic tape or rotating memory unit, tailored to the hardware and software requirements of the installation.

System preparation is controlled by SPP control directives. These directives include a description of the devices to be used for the system preparation, the devices to be included in the generated MOS system file, the system preparation functions to be executed, and the parameters for the system preparation functions.

Status and error messages are given in Section 13.

## CONTROL DIRECTIVES

Control directives can be presented to the SPP from the SI in any order, except that the END directive must always be last. END signals the end of the SPP control directives and causes the program to begin generating the MOS system file on the PO.

SPP directives have the following specifications:

- a. General form:

name,p(1),p(2),...,p(n)

where name is the directive name and p(1),p(2),...,p(n) is a parameter string separated by commas.

- b. Directives have up to 72 characters and begin in the first character position of the record (card).
- c. Blank records are ignored.
- d. Embedded blanks are allowed within a field.
- e. Numeric fields can be signed or unsigned octal or decimal integers. Octal numbers begin with a zero and decimal numbers do not.
- f. Fields smaller than the maximum are left-justified and blank-filled.

## DIRECTIVES

**ADD,name(1),name(2),...,name(n)**

ADD adds a program or group of programs to the system file. The programs contained on the BI are added selectively to the system file after each program specified by ADD. Each name is the name of a program in the installation system library (ISL).

In processing ADD, the SPP copies from the ISL to the system file until it has copied the program specified by a name in ADD. Then, the SPP types the message:

**ADD name**

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

- Y** Add the program contained on the BI, copy it to the system file, and return for another directive.
- N** The additions are complete. Resume copying to the next name. N also types the message:

#### **END ADDITIONS**

**Note:** The ADD control directive cannot be used to add a new program within an ABS-ENDABS program module.

#### **DELETE,name(1),name(2),...,name(n)**

DELETE deletes the specified program or programs from the ISL when the system file is produced. Each name is the name of a program in the ISL.

In processing DELETE, the SPP copies from the ISL to the system file all programs except those named in DELETE.

#### **REPLACE,name(1),name(2),...,name(n)**

REPLACE replaces the specified program or programs from the ISL with new programs contained on the BI. Each name is the name of a program in the ISL.

In processing REPLACE, the SPP copies from the ISL to the system file until it has copied the program preceding the program specified by name in REPLACE. The SPP skips the specified program on the ISL and types the message:

#### **REPLACE name**

on the Teletype (and LO, if different). The program then waits for one of the following control directives from the Teletype:

- Y** Add the program contained on the BI, copy it to the system file, and return for another directive.
- N** The replacements are complete. Resume copying to the next name. N also types the message:

## END REPLACEMENTS

### END,p(1),p(2)

END indicates to the SPP that there are no more system preparation control directives to process. Its parameters are:

- |          |                                                                                                    |
|----------|----------------------------------------------------------------------------------------------------|
| <b>L</b> | List the ISL on the LO.                                                                            |
| <b>N</b> | Do not list the ISL nor verify and list the new system file.                                       |
| <b>V</b> | Verify the new system file.                                                                        |
| blank    | Do not list the ISL, but verify and list the new system file (in this case, no comma follows END). |

### DATE,xxxxxxx

DATE sets up to eight alphanumeric characters (e.g., a date) in the page title of the list output. If DATE is not used, no date is printed.

### COMP,p(1),p(2),p(3),p(4),p(5),p(6),p(7)

COMP defines the MOS computer system configuration as follows:

- |             |                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>p(1)</b> | Specifies the computer. If it is a 620/a or 622/a, p(1) is A. If it is a 620/i, 620/L, 622/i, R620/i, R622/i, or 620/f, P(1) is I or unspecified.                                                                                                                                                                                                                                                                  |
| <b>p(2)</b> | Specifies the highest available memory address to be used for system preparation and the MOS system being prepared. On a system with 12K or larger memory, 025777 is assumed if the value is unspecified or less than 025777. On a system with 8K of memory, 017777 is assumed if the value is unspecified or less than 017777.                                                                                    |
| <b>p(3)</b> | Specifies the number (0, 1, 2, 3, or 4) of buffer interface controllers (BICs) available in the system. If the value is unspecified or larger than four, zero is assumed. The SPP uses this parameter to supply the proper set of BIC library subroutines. If no BIC is available, dummy BIC library subroutines are entered in the system file. Otherwise, the program enters the actual BIC library subroutines. |

- p(4)** Specifies the relocation bias set by the executive for the first program to be loaded by the loader if not specified in /LOAD (Section 2). A value of P(4)-1 is specified or less than the value of p(5), 04000 is assumed.
- p(5)** Specifies the base address set by the executive to be used by the loader for the direct literal pool if not specified in /LOAD (Section 2). If the value is unspecified if p(5) is either not specified, larger than 03777, or smaller than p(6).
- p(6)** Specifies the base address set by the executive to be used by the loader for the indirect address pool if not specified in /LOAD (Section 2). A value of 010 is assumed if p(6) is either not specified, smaller than 010, larger than 0777, or larger than p(5).
- p(7)** Specifies the number of logical units in MOS. If it is unspecified, less than 14, or greater than 255, a value of 14 is assumed.

Commas must be present for unspecified parameters unless all parameters are unspecified. Thus, to define p(2) and p(7) only, input

```
COMP, , 027777, , , , 25
```

and to set all parameters to their default values, input

```
COMP
```

```
IODEV,p(1),p(2),p(3)
```

IODEV adds or modifies entries in the MOS logical and physical unit tables as follows:

- p(1)** Specifies a four-character alphanumeric I/O device name. If /ASSIGN (Section 2) references this name, the SPP modifies the logical unit table to point to the physical unit.
- p(2)** Specifies the second and third characters of the three-character subroutine entry name of the I/O driver. The SPP stores these characters as the second two characters of the I/O driver in the physical unit table (the first character is always the dollar sign).

**P(3)** Specifies the relative position the I/O driver occupies in the physical unit table. The SPP enters the name of the I/O driver in the physical unit table in the order specified by p(3). If p(3) is omitted, the I/O driver is added to the physical unit table at the bottom.

Commas must be present for unspecified parameters unless all parameters are unspecified. Thus, to define p(1) and p(2) only, input

**IODEV,MTU3,VW,**

**EQUIP,p(1),p(2),p(3),...,p(n)**

EQUIP specifies the I/O devices to be included in the preparation of the system file. The SPP uses the parameters to construct the MOS logical and physical unit tables, and to select the proper drivers from the ISL when building the system file.

EQUIP parameters p(1) through p(n) are:

|                  |                                                                            |
|------------------|----------------------------------------------------------------------------|
| <b>TYcu</b>      | Teletype keyboard/page printer                                             |
| <b>TRcu</b>      | Teletype paper tape reader                                                 |
| <b>TPcu</b>      | Teletype paper tape punch                                                  |
| <b>CRcu</b>      | Card reader                                                                |
| <b>CPcu</b>      | Card punch                                                                 |
| <b>LPcu</b>      | Line printer                                                               |
| <b>PTcu</b>      | High-speed paper tape reader and punch                                     |
| <b>PTcu(R)</b>   | High-speed paper tape reader only                                          |
| <b>PTcu(P)</b>   | High-speed paper tape punch only                                           |
| <b>MTcu(x)</b>   | Magnetic tape unit to be connected to BIC number x                         |
| <b>MTcu</b>      | Magnetic tape unit                                                         |
| <b>DRcu(x,y)</b> | Drum memory unit to be connected to BIC number x; with y sectors allocated |
| <b>DKcu(x,y)</b> | Disc memory unit to be connected to BIC number x; with y sectors allocated |

where

|            |                                                             |
|------------|-------------------------------------------------------------|
| <b>cu</b>  | specifies controller and unit number; if omitted, assume 00 |
| <b>P,R</b> | are entered as key words (symbols)                          |
| <b>X,Y</b> | are numeric variables                                       |

**Drum Partitioning.** Because of the low access time of the drum memory unit, MOS can partition the drum into from one to ten virtual units. Each virtual unit can function as a separate logical unit (except that the drum can perform only one operation at a time). Beginning- and end-of-device sector addresses and a current address pointer for each unit are kept by MOS in the resident constant area. End-of-file marks are recorded on the first ten sectors of the drum.

The ten possible drum units are designated DR00 through DR09. EQUIP partitions the drum into the desired number of virtual units. A particular virtual unit is incorporated into the system if it is designated in an EQUIP parameter of the form **DRcu (x,y)**. Since x specifies the BIC, all drum units have the same x value. The number of sectors allocated to that virtual unit is given by y. If DR00 is designated as logical unit SF, y is the number of sectors assigned to DR00 in addition to those used by MOS. The value of x and y can be octal or decimal, but, if the former, it has a leading zero.

- If DR00 is specified, it occupies sectors 012 to 012 + y if it is not the SF. If DR00 is SF, it occupies sectors 012 to 012 + y + k, where k is the number of sectors in the system file.
- Other specified virtual units begin at the end of the previous unit and occupy the next y sectors or the rest of the drum, whichever is smaller.

Since the size of the system file differs for each configuration, the number of sectors it occupies on DR00 is not known at the beginning of system preparation. The SPP, therefore, lists the sector allocation for each virtual unit on LO at the end of system preparation in the form:

#### DRUM ALLOCATION

|             |               |               |
|-------------|---------------|---------------|
| <b>DR00</b> | <b>000012</b> | <b>xxxxxx</b> |
| <b>DR01</b> | <b>yyyyyy</b> | <b>xxxxxx</b> |
| .           | .             | .             |
| .           | .             | .             |
| .           | .             | .             |
| <b>DR09</b> | <b>yyyyyy</b> | <b>xxxxxx</b> |

where

|               |                             |
|---------------|-----------------------------|
| <b>xxxxxx</b> | is the last sector address  |
| <b>yyyyyy</b> | is the first sector address |



All addresses are octal. Typical system files occupy about 03500 sectors.

If DR00 contains the MOS system file, the drum should be partitioned into at least two virtual units. The second unit is for intermediate storage of the source statements during assemblies and for other utility tasks. DR00 cannot contain both the MOS system file and assembly source statements.

**Disc Partitioning.** A disc MOS system can have one or two disc units. MOS can partition each disc unit into from one to ten virtual units. Each virtual unit can function as a separate logical unit (with the obvious exception that two virtual units on the same physical unit cannot function simultaneously). Beginning- and end-of-device sector addresses and a current address pointer for each virtual unit are maintained by the system in the resident monitor.

The designations of the 30 possible virtual units, and the physical unit and controller corresponding to each, are shown below. EQUIP partitions the disc into the desired number of virtual units.

| Controller | Disc Unit | Virtual Unit |
|------------|-----------|--------------|
| 0          | 0         | DK00-DK09    |
| 0          | 1         | DK10-DK19    |
| 1          | 0         | DK40-DK49    |

A particular disc unit is incorporated into MOS if (and only if) one or more corresponding virtual unit designations appear in an EQUIP parameter of the form:

$$DKcu(x,y)$$

Since x specifies the BIC, DK00 through DK19 have the same n value. Similarly, DK40 through DK49 have the same n value.

The number of sectors to be allocated to that virtual unit is given by y. The value of y can be octal or decimal, but, if the former, it has a leading zero.

- a. If DK00 is specified, it begins in sector 0 of the corresponding disc unit and occupies the next y sectors.
- b. Each additional virtual unit specified for the same disc unit begins at the end of the previous virtual unit and occupies the number of sectors specified.
- c. If the total number of sectors specified for all the virtual units is less than the total number on the disc, the last virtual unit is expanded to fill the rest of the disc. If the total number of sectors exceeds the total number on the disc, the first virtual unit to exceed the disc size is truncated to the capacity of the disc and all additional virtual units are zero sectors long.

- d. If DK00 is SF, it begins in sector 0 and occupies the next  $y + k$  sectors, where  $k$  is the number of sectors required by the MOS system file.

Since the size of the system file differs for each configuration, the number of sectors it occupies on DK00 is unknown at the beginning of system preparation. The SPP, therefore, lists the sector allocation for each virtual unit on LO at the end of system preparation in the form:

```

DISC 0 ALLOCATION

DK00      000000      xxxxxx
DK01      YYYYYY      xxxxxx
.          .          .
.          .          .
.          .          .
DK09      YYYYYY      xxxxxx

DISC 1 ALLOCATION

DK10      000000      xxxxxx
DK11      YYYYYY      xxxxxx
.          .          .
.          .          .
.          .          .
DK19      YYYYYY      xxxxxx

```

where

**xxxxxx**      is the last sector address  
**yyyyyy**      is the first sector address

All address are octal. Typical system files occupy about 03500 sectors.

If DK00 contains the MOS system file, the first disc unit is partitioned into at least two virtual units. The second unit is for intermediate storage of the source statements during assemblies, and for other utility tasks. DK00 cannot contain both the MOS system file and assembly source statements.

**ASSIGN,  $l(1) = r(1), l(2) = r(2), \dots, l(n) = r(n)$**

Assign equates and assigns particular logical units to specific physical I/O devices. Execution of this directive decodes the parameter string and alters the logical unit table as specified by the parameter.

The parameters can be logical unit numbers, logical unit names, or physical unit names (Figure 3-1). In each parameter pair (i.e., each  $l(n) = r(n)$ ), the left parameter,  $l(n)$ , is a

## system preparation program

logical unit number or name, and the right parameter,  $r(n)$ , is a logical unit number or name or a physical device name.

In any case, the logical unit to the left of the equal sign is assigned to the unit/device to the right.

If  $r$  is a physical device, the  $l$  entry in the logical unit table is altered so that it points to the physical device driver specified by  $r$ . Thereafter, all I/O operations referencing  $l$  are directed to the physical device specified by  $r$ .

If  $r$  is a logical unit number or name,  $l$  is made equivalent to  $r$  and is assigned to the same physical device as  $r$ . However, if  $r$  is reassigned later to a new physical device,  $l$  no longer has an equivalent assignment.

The SPP makes default assignments for logical units not assigned to physical units. These assignments are a function of the physical devices included in the system. The SPP nominally assigns each logical unit the number of the highest peripheral device listed in Figure 8-1.

SF is always assigned to the same peripheral device that was used as PO in system preparation. SS is always assigned to the same logical unit as PO.

The first scratch unit is assigned to the first available device, the second scratch unit to the next available device, etc. If the end of the table is encountered, the table is rescanned from the top rather than assigning the logical unit to DUM. If no device is available, scratch units are assigned to DUM.

### **ABS,p(1),p(2),p(3),p(4),p(5)**

ABS generates an absolute module on the system file. The input is an object module generated by assembly or compilation, preceded by the ABS. When the SPP encounters the ABS, the first object module that follows is converted into an absolute module and put on the MOS system file. Any following subprograms are only converted into the absolute module if referenced by the first (or a previous) program. No program or subprogram can contain an instruction reference to an externally defined literal.

The ABS parameters are:

- p(1)** An eight-character ASCII program identification name. The SPP stores this name in the identification block of the beginning absolute loader text record. If  $p(1)$  is omitted, the program generates a unique program identification name of the form XXnnnn for the program (where nnnn is a decimal number beginning at 0001 for the first program).
- p(2)** An eight-character creation date. The SPP stores  $p(2)$  in the date block of the beginning absolute loader text record.

| SI   | SO   | PI   | LO   | BI   | BO   | GO   | PO   | S1,S2,S3,S4 |
|------|------|------|------|------|------|------|------|-------------|
| TY00 | TY00 | CR00 | LP00 | CR00 | CP00 | MT02 | MT01 | MT01        |
| TY10 | TY10 | CR10 | LP10 | CR10 | CP10 | DR02 | DR01 | DR01        |
| CR00 | DUM  | PT00 | TY00 | PT00 | PT00 | MT03 | MT02 | MT02        |
| CR10 |      | PT10 | TY10 | PT10 | PT10 | DR03 | DR02 | DR02        |
| DUM  |      | TR00 | DUM  | TR00 | TP00 | MT10 | MT03 | MT03        |
|      |      | TR10 |      | TR10 | TP10 | DR04 | DR03 | DR03        |
|      |      | DUM  |      | DUM  | DUM  | MT11 | MT10 | MT10        |
|      |      |      |      |      |      | DR05 | DR04 | DR04        |
|      |      |      |      |      |      | MT12 | MT11 | MT11        |
|      |      |      |      |      |      | DR06 | DR05 | DR05        |
|      |      |      |      |      |      | MT13 | MT12 | MT12        |
|      |      |      |      |      |      | DR07 | DR06 | DR06        |
|      |      |      |      |      |      | MT20 | MT13 | MT13        |
|      |      |      |      |      |      | DR08 | DR07 | DR07        |
|      |      |      |      |      |      | MT21 | MT20 | MT20        |
|      |      |      |      |      |      | DR09 | DR08 | DR08        |
|      |      |      |      |      |      | MT22 | MT21 | MT21        |
|      |      |      |      |      |      | MT23 | DR09 | DR09        |
|      |      |      |      |      |      | MT30 | MT22 | MT22        |
|      |      |      |      |      |      | MT31 | MT23 | MT23        |
|      |      |      |      |      |      | MT32 | MT30 | MT30        |
|      |      |      |      |      |      | MT33 | MT31 | MT31        |
|      |      |      |      |      |      | DUM  | MT32 | MT32        |
|      |      |      |      |      |      |      | MT33 | MT33        |
|      |      |      |      |      |      |      | MT00 |             |
|      |      |      |      |      |      |      | DUM  |             |

Figure 8-1. Valid SPP Logical Unit Assignments

- p(3)** A program relocation bias. The SPP uses this value to define the start of the first of the generated programs. If p(3) is omitted, the SPP uses the value of p(4) in COMP. If the p(3) is a -1, the SPP uses the current value of p(2) in COMP minus the size of this program as the relocation bias.
- p(4)** The base address of the direct literal pool. If the p(4) parameter is omitted, larger than 03777, larger than p(3), or smaller than p(5), the SPP uses the value of p(5) in COMP.
- p(5)** Specifies the base address of the indirect address pointers. If p(5) is omitted, smaller than 010, larger than 0777, or larger than p(4), SPP uses the value specified by COMP.

### **ENDABS**

ENDABS terminates ABS processing. It follows the object modules input with the ABS.

# INSTALLATION SYSTEM LIBRARY ORGANIZATION

The installation system library (ISL) is the primary input to the SPP. The order of the sections in the ISL and the programs within the sections must be maintained for proper MOS system file preparation and operation. The ISL sections are:

- a. System preparation
- b. System processor
- c. System library

Comment records are alphameric records with a blank in the first character position. They are between, but not within, programs.

## SYSTEM PREPARATION SECTION

This section contains:

- a. Loader for the system preparation program
- b. System preparation program part 1
- c. System preparation program part 2

## SYSTEM PROCESSOR SECTION

This section contains:

- a. Resident system configuration block part 1 (resident monitor)
- b. Resident system configuration block part 2 (dump)
- c. Resident system configuration block part 3 (I/O control)
- d. Resident system configuration block part 4 (executive)
- e. Loader
- f. Loader system file I/O drivers
- g. Loader map subroutine

## **system preparation program**

- h. Loader list output I/O drivers
- i. Loader binary input I/O drivers
- j. DAS MR assembler
- k. FORTRAN IV compiler
- l. Debugging program
- m. Concordance program
- n. System maintenance program
- o. File editing program
- p. Input/output drivers

## **SYSTEM LIBRARY SECTION**

This section comprises the FORTRAN IV run-time library and programs that perform utility functions. Utility or often-used object programs can be added and executed with /ULOAD (section 2). The final programs in this section are the I/O drivers. All user programs precede the I/O drivers.

## OPERATING PROCEDURES

To prepare a MOS system:

- a. Load the system preparation program (SPP) by entering the proper bootstrap program into the computer memory.
- b. Assign the peripheral devices for use by the SPP.
- c. Supply the control directives to define a MOS system file.

## LOADING

Depending on the peripheral device used to read the SPP, one of the following initialization procedures applies:

- a. Card reader
  - (1) Turn on the card reader.
  - (2) Place two blank cards after the last control-directive card of the ISL deck.
  - (3) Place the ISL deck in the card hopper.
  - (4) Press clear and start.
- b. 33/35 ASR Teletype
  - (1) Turn on the Teletype
  - (2) Place Teletype in off-line mode and simultaneously press the CONTROL and D, then CONTROL and T, and finally the CONTROL and Q keys.
  - (3) Position the system preparation loader program paper tape in the reader with the first binary frame at the reading station. Close the reading gate.
  - (4) Set the reader control level to STOP, and the Teletype on-line.
- c. High-speed paper tape reader
  - (1) Turn on the paper tape reader.
  - (2) Position the system preparation loader program paper tape in the reader with the first binary frame at the reading station. Close the reading gate.
  - (3) Set the LOAD/RUN switch to RUN.
- d. Magnetic tape unit
  - (1) Turn on the magnetic tape unit.
  - (2) Mount the ISL magnetic tape.
  - (3) Position the magnetic tape to the load point.
  - (4) Ready the magnetic tape unit so it can be used by the computer.



Enter the appropriate bootstrap loading routine (Figures 8-2 and 8-3) in memory as follows:

- a. Turn on REPEAT (and position STEP/RUN to STEP for 620/f).
- b. Enter instruction 054000 (store A relative to P) in the instruction register.
- c. Reset the P register.
- d. Enter a bootstrap instruction in the A register.
- e. Press STEP (START for 620/f).
- f. Reset the A register.
- g. Repeat steps d, e, and f for each bootstrap instruction.

Initiate the bootstrap from the peripheral device as follows:

- a. To initiate loading from the card reader, high-speed paper tape reader, or magnetic device, reset the A, B, X, P, and U (I for 620/f) registers; then, press SYSTEM RESET and RUN (for 620/f press RESET, position STEP/RUN to RUN, and press START).

The system preparation loader and ISL are separate paper tapes. The ISL must be mounted when the computer goes to STEP after reading the loader.

- b. To initiate loading from the Teletype, follow step a; then, set the reader control level to RUN. (Start position on ASR 33.)

#### **NOTE**

If an error occurs while loading the system preparation modules, the computer goes to the STEP mode with the instruction register = 0777 and  $A = B = X = -1$ . Recovery is made by repositioning the last record read at the read station and press RUN (START for 620/f). For magnetic tape, the repositioning is automatic.

## **ASSIGNMENT**

At the end of a successful loading, the Teletype makes five requests for peripheral device assignments to be used by the SPP. The form of these requests is:

| Address | Magnetic<br>Tape<br>Controller 0<br>Unit 0 | Magnetic<br>Tape<br>Controller 0<br>Unit 1 | Magnetic<br>Tape<br>Controller 1<br>Unit 0 | 33/35<br>ASR<br>Teletype | High-Speed<br>Paper<br>Tape<br>Reader | Card<br>Reader |
|---------|--------------------------------------------|--------------------------------------------|--------------------------------------------|--------------------------|---------------------------------------|----------------|
| 00000   | 104110                                     | 104210                                     | 104111                                     | 102601                   | 100537                                | 100230         |
| 00001   | 101210                                     | 101210                                     | 101211                                     | 030011                   | 030011                                | 101130         |
| 00002   | 000005                                     | 000005                                     | 000005                                     | 005101                   | 005101                                | 000007         |
| 00003   | 001000                                     | 001000                                     | 001000                                     | 101201                   | 101537                                | 101630         |
| 00004   | 000001                                     | 000001                                     | 000001                                     | 000007                   | 000007                                | 0xx400         |
| 00005   | 030020                                     | 030020                                     | 030020                                     | 001000                   | 001000                                | 001000         |
| 00006   | 100010                                     | 100010                                     | 100011                                     | 000003                   | 000003                                | 000001         |
| 00007   | 102510                                     | 102510                                     | 102511                                     | 102601                   | 102637                                | 102230         |
| 00010   | 055000                                     | 055000                                     | 055000                                     | 001020                   | 001020                                | 030024         |
| 00011   | 005144                                     | 005144                                     | 005144                                     | 0xx400                   | 0xx400                                | 004244         |
| 00012   | 101110                                     | 101110                                     | 101111                                     | 004050                   | 004050                                | 004344         |
| 00013   | 000007                                     | 000007                                     | 000007                                     | 004002                   | 004002                                | 004444         |
| 00014   | 101210                                     | 101210                                     | 101211                                     | 004446                   | 004446                                | 060030         |
| 00015   | 0xx401                                     | 0xx401                                     | 0xx401                                     | 001020                   | 001020                                | 020027         |
| 00016   | 001000                                     | 001000                                     | 001000                                     | 000003                   | 000003                                | 004142         |
| 00017   | 000012                                     | 000012                                     | 000012                                     | 055000                   | 055000                                | 056000         |
| 00020   | 0xx400                                     | 0xx400                                     | 0xx400                                     | 005144                   | 005144                                | 005344         |
| 00021   |                                            |                                            |                                            | 001000                   | 001000                                | 040027         |
| 00022   |                                            |                                            |                                            | 000002                   | 000002                                | 020030         |
| 00023   |                                            |                                            |                                            |                          |                                       | 001040         |
| 00024   |                                            |                                            |                                            |                          |                                       | 000003         |
| 00025   |                                            |                                            |                                            |                          |                                       | 001000         |
| 00026   |                                            |                                            |                                            |                          |                                       | 000011         |
| 00027   |                                            |                                            |                                            |                          |                                       | yy5777         |

xx = 17 for 8K systems; xx = 25 for 12K or larger systems  
yy = 07 for 8K systems; yy = 12 for 12K or larger systems

Figure 8-2. 620 Bootstrap Loading Routines

| Address | Magnetic<br>Tape<br>Controller 0<br>Unit 0 | Magnetic<br>Tape<br>Controller 0<br>Unit 1 | Magnetic<br>Tape<br>Controller 1<br>Unit 0 | 33/35<br>ASR<br>Teletype | High-Speed<br>Paper<br>Tape<br>Reader | Card<br>Reader |
|---------|--------------------------------------------|--------------------------------------------|--------------------------------------------|--------------------------|---------------------------------------|----------------|
| 00000   | 104110                                     | 104210                                     | 104111                                     | 102601                   | 100537                                | 100230         |
| 00001   | 101210                                     | 101210                                     | 101211                                     | 030011                   | 030011                                | 001000         |
| 00002   | 000005                                     | 000005                                     | 000005                                     | 005101                   | 005101                                | 000006         |
| 00003   | 001000                                     | 001000                                     | 001000                                     | 101201                   | 101537                                | 0xx377         |
| 00004   | 000001                                     | 000001                                     | 000001                                     | 000007                   | 000007                                | 040003         |
| 00005   | 030020                                     | 030020                                     | 030020                                     | 001000                   | 001000                                | 067003         |
| 00006   | 100010                                     | 100010                                     | 100011                                     | 000003                   | 000003                                | 040003         |
| 00007   | 102510                                     | 102510                                     | 102511                                     | 102601                   | 102637                                | 006010         |
| 00010   | 055000                                     | 055000                                     | 055000                                     | 001020                   | 001020                                | 001036         |
| 00011   | 005144                                     | 005144                                     | 005144                                     | 0xx400                   | 0xx400                                | 050022         |
| 00012   | 101110                                     | 101110                                     | 101111                                     | 004052                   | 004052                                | 005007         |
| 00013   | 000007                                     | 000007                                     | 000007                                     | 004002                   | 004002                                | 101630         |
| 00014   | 101210                                     | 101210                                     | 101211                                     | 004446                   | 004446                                | 0xx400         |
| 00015   | 0xx401                                     | 0xx401                                     | 0xx401                                     | 001020                   | 001020                                | 101130         |
| 00016   | 001000                                     | 001000                                     | 001000                                     | 000003                   | 000003                                | 000021         |
| 00017   | 000012                                     | 000012                                     | 000012                                     | 055000                   | 055000                                | 001000         |
| 00020   | 0xx400                                     | 0xx400                                     | 0xx400                                     | 005144                   | 005144                                | 000013         |
| 00021   |                                            |                                            |                                            | 001000                   | 001000                                | 102230         |
| 00022   |                                            |                                            |                                            | 000002                   | 000002                                | 001036         |
| 00023   |                                            |                                            |                                            |                          |                                       | 000004         |
| 00024   |                                            |                                            |                                            |                          |                                       | 004454         |
| 00025   |                                            |                                            |                                            |                          |                                       | 057003         |
| 00026   |                                            |                                            |                                            |                          |                                       | 040022         |
| 00027   |                                            |                                            |                                            |                          |                                       | 001000         |
| 00030   |                                            |                                            |                                            |                          |                                       | 000013         |

xx = 17 for 8K systems; xx = 25 for 12K or larger systems

Figure 8-3. 622 Bootstrap Loading Routines

**ENTER DEVICE NAME FOR xx**

where xx denotes PI, PO, LO, BI, and SI, respectively, in the five requests. In response to each request, type the name of a peripheral device followed by a carriage return. The specified peripheral device is assigned the corresponding logical function during the system preparation process.

Figure 8-4 gives the function of each logical unit during system preparation and lists acceptable peripheral device assignments for each logical unit name. To assign the peripheral device a default assignment, type a carriage return.

After completing peripheral device assignments for system preparation, the message:

**BEGIN SYSTEM PREPARATION**

is output. The SPP can then accept system preparation control directives defining the MOS system file to be generated.

**DISC FORMATTING**

When preparing an MOS system for a 620-40 disc, a disc-formatting routine is loaded by the SPP loader. This routine is loaded prior to the SPP.

At the end of a successful loading, the Teletype makes four requests for disc formatting information. These requests are for the BIC hardware address, controller hardware address, disc unit number, and the disc size (in cylinders). Each input can be either a decimal or octal number, but, if the latter, it has a leading zero. Each request is terminated by a carriage return.

After the fourth request, the routine formats the disc. When formatting is complete, formatting information for another disc is requested. When all discs have been formatted, answer the request for the BIC hardware address by pressing the CONTROL and BELL keys on the Teletype. The loader then loads the next segment of the SPP and proceeds as described in Section 4.

**SYSTEM VERIFICATION AND COMPLETION**

Upon completion of the system preparation, the system file is read and verified to ensure that no errors occurred. Verification consists of reading the new system file and checking such items as checksums, sequence numbers, and record formats. A listing is also generated on the LO (figure 8-5). If verification is not desired, supply the parameter N on the END control directive. After the system preparation is complete, the program outputs the message:

**MOS SYSTEM READY**

It then loads the resident monitor into memory by executing the appropriate bootstrap of the MOS system file and types \*\* on the Teletype.

| Unit | Function                                                                      | Assignments                                                       | Default |
|------|-------------------------------------------------------------------------------|-------------------------------------------------------------------|---------|
| PI   | Contains the ISL                                                              | TR00*****<br>PT00<br>CR00*<br>MT01*****<br>MT10*****<br>MT00***** | TR00    |
| PO   | Contains the MOS system                                                       | MT00*****<br>DR00**<br>DK00***                                    | MT00    |
| LO   | Lists all SPP control directives and the ISL or MOS system file, if requested | TY00<br>LP00                                                      | TY00    |
| BI   | Inputs any additional programs not contained on the ISL                       | TR00<br>PT00<br>CR00*                                             | TR00    |
| SI   | Inputs all SPP control directives                                             | TY00<br>TR00<br>PT00<br>CR00*                                     | TY00    |

\*Only on systems with 12K or larger memory.

\*\*DR00 is acceptable only on the drum and magnetic tape MOS.

\*\*\*DK00 is acceptable only on the disc MOS.

\*\*\*\*MT00 is acceptable on systems with 12K or larger memory or any drum and magnetic tape MOS.

\*\*\*\*\*When TR00 is specified, TP00 must also be specified within the EQUIP directive parameter.

**Figure 8-4. Logical Unit Functions**

PAGE 1

```
BEGIN SYSTEM PREPARATION
DATE,01/20/71
COMP,I,025777,1,0500,0477,010,14
EQUIP,MT00,MT10,TY,TR,TP,PT,CR,LP
EQUIP,DK00(022,0),DK01(022,02000),DK02(022,01000),DK03(022,01000)
END,L
```

Figure 8-5. System Preparation List Output (1 of 3)

PAGE 2 01/20/71

| ID                        | DATE     | SIZE         | ENTRY NAMES | EXTERNAL NAMES |
|---------------------------|----------|--------------|-------------|----------------|
| A BUNTSTRP 01/20/71 00442 |          |              |             |                |
| ABS,RSCR1DK               | 05/12/70 | -1,01000,010 |             |                |
| SPUR                      | 25006    |              |             |                |
| SLUB                      | 25037    |              |             |                |
| CYLND4                    | 25766    |              |             |                |
| CYLND1                    | 25766    |              |             |                |
| CYLND0                    | 25740    |              |             |                |
| SXEQ                      | 25066    |              |             |                |
| SUCB4                     | 25741    |              |             |                |
| SUCB1                     | 25741    |              |             |                |
| SUCB0                     | 25713    |              |             |                |
| STTL                      | 25072    |              |             |                |
| SPDC                      | 25776    |              |             |                |
| SREN                      | 25505    |              |             |                |
| SPED                      | 25070    |              |             |                |
| SPCW                      | 25056    |              |             |                |
| SMPR                      | 25146    |              |             |                |
| SLCB                      | 25067    |              |             |                |
| SLBF                      | 00400    |              |             |                |
| SLAX                      | 25255    |              |             |                |
| SLAB                      | 25270    |              |             |                |
| SJCW                      | 25057    |              |             |                |
| SPRM                      | 25102    |              |             |                |
| SECH                      | 25071    |              |             |                |
| SDAT                      | 25076    |              |             |                |
| SBUF                      | 25617    |              |             |                |
| [SIAP]                    | 00010    |              |             |                |
| [SLIT]                    | 01000    |              |             |                |
| [SPED]                    | 25055    |              |             |                |
| ABS,RSCR1MT               | 01/11/70 | -1,01000,010 |             |                |
| ENDABS                    |          |              |             |                |
| ABS,RSCR2TY               | 01/11/70 | -1,01000,010 |             |                |
| ENDABS                    |          |              |             |                |
| ABS,RSCR2LP               | 01/11/70 | -1,01000,010 |             |                |
| SPUR                      | 25006    |              |             |                |
| SLUB                      | 25037    |              |             |                |
| CYLND4                    | 25766    |              |             |                |
| CYLND1                    | 25766    |              |             |                |
| CYLND0                    | 25740    |              |             |                |
| SXEQ                      | 25066    |              |             |                |

Note: If a slash appears to the left of a number field, the designated program is required by the system preparation program but omitted from the ISL.

Figure 8-5. System Preparation List Output (2 of 3)

PAGE 9 01/20/71

| IC      | DATE | SIZE  | ENTRY NAMES | EXTERNAL NAMES      |
|---------|------|-------|-------------|---------------------|
|         |      |       | \$LBF \$LAX |                     |
|         |      |       | \$LAB \$JCH |                     |
|         |      |       | \$IOK \$IAP |                     |
|         |      |       | \$PRM \$ECN |                     |
|         |      |       | \$DAT \$CDR |                     |
|         |      |       | \$COM \$BUF |                     |
|         |      |       | \$ALG       |                     |
| R MAPS  |      | 00123 | MAPS        | LCBS IOCS \$BUF     |
| R \$00  |      | 00022 | \$00        | MRS MT\$0 MRS MSFS  |
| *R \$01 |      | 00022 | \$01        | MRS MRD\$ MRS MCK\$ |
| *R \$02 |      | 00022 | \$02        | MRS MT\$0 MRS MSFS  |
| *R \$03 |      | 00022 | \$03        | MRS MRD\$ MRS MCK\$ |
| R \$04  |      | 00022 | \$04        | MRS MT\$1 MRS MSFS  |
| *R \$05 |      | 00022 | \$05        | MRS MRD\$ MRS MCK\$ |
| *R \$06 |      | 00022 | \$06        | MRS MT\$1 MRS MSFS  |
| *R \$07 |      | 00022 | \$07        | MRS MRD\$ MRS MCK\$ |
| *R \$08 |      | 00022 | \$08        | MRS MT\$1 MRS MSFS  |
| *R \$09 |      | 00022 | \$09        | MRS MRD\$ MRS MCK\$ |
| *R \$0A |      | 00022 | \$0A        | MRS MT\$2 MRS MSFS  |
| *R \$0B |      | 00022 | \$0B        | MRS MRD\$ MRS MCK\$ |
| *R \$0C |      | 00022 | \$0C        | MRS MT\$2 MRS MSFS  |
| *R \$0D |      | 00022 | \$0D        | MRS MRD\$ MRS MCK\$ |
| *R \$0E |      | 00022 | \$0E        | MRS MT\$3 MRS MSFS  |
| *R \$0F |      | 00022 | \$0F        | MRS MRD\$ MRS MCK\$ |

Note: An asterisk preceding a line indicates that the designated program is contained in the ISL but omitted from the created System File.

Figure 8-5. System Preparation List Output (3 of 3)



## system preparation program

If the system file is on a rotating memory unit, the virtual unit allocation table is listed on SO followed by the message:

DISC MOS SYSTEM READY

## EXAMPLES

### Problem 1

Prepare an MOS system file for a 620/622 computer system having one magnetic tape unit, one Teletype unit, and 8K of core memory. Make the following logical unit assignments: SF = MT00, PI = TR00, SS = MT00, BI = TR00, SO = TY00, LO = TY00, and BO = TP00. Set the default values of \$PGM to 03000, \$LIT to 02777, and \$IAP to 010. During system preparation, do not list the ISL, but verify and list the MOS system file.

### Procedure:

1. Key in the Teletype bootstrap loader and enter the SPP through the Teletype paper tape reader.
2. Assign logical units for use by the SPP as follows:

```
PI = TR00
PO = MT00
LO = TY00
BI = TR00
SI = TY00
```

3. Mount the ISL on the Teletype reader (PI) and mount a scratch magnetic tape with a write-ring on the magnetic tape transport (PO).
4. Respond to the **BEGIN SYSTEM PREPARATION** message with the following control directives on the Teletype keyboard (SI):

```
COMP, I, 017777, 0, 03000, 02777, 010, 14
DATE, 11/07/69          (optional)
EQUIP, TY, MT, TR, TP
END
```

## Problem 2

Prepare an MOS system file for a 620/622 computer system having four standard magnetic tape units connected to one controller, one special magnetic tape unit connected to a different controller for which the user supplies the I/O driver, one Teletype unit, one line printer, one card reader, one high-speed paper tape reader/punch unit, and 16K of core memory. The special magnetic tape unit is designated MM10 and its I/O driver has the entry name \$OU. Make the following logical unit assignments: SF = MT00, PO = MT01, SS = MT01, GO = MT02, SI = TY00, PI = CR00, LO = LP00, BI = PT00, BO = PT00, S1 = MT03, and LUN 15 = MM10 (special magnetic tape unit). Set the default values of \$PGM to 04000, \$LIT to 03777, and \$IAP to 030. During the system preparation process, replace the FORTRAN compiler with a new version and verify and list both the ISL and the MOS system file.

### Procedure:

1. Key in the magnetic tape unit 2 bootstrap and enter the SPP through MT01.
2. Assign logical units for use by the SPP as follows:

```
PI = MT01
PO = MT00
LO = LP00
BI = PT00
SI = TY00
```

3. Mount the ISL on MT01 (PI) and mount a magnetic tape on MT00 (PO).
4. Respond to the **BEGIN SYSTEM PREPARATION** message by typing the following control directives on the Teletype (SI):

```
COMP,I,037777,0,04000,03777,030,15
DATE,11/07/69          (optional)
IODEV,MM10,OU,5
EQUIP,TY,CR,LP,PT,TR,TP
EQUIP,MT00,MT01,MT02,MT03,MM10
ADD,$03
REPLACE FORTRAN
ASSIGN,BI=PT00,BO=BI
assign, S1 = MT03, 15 = MM10
```

```
END,L
```

5. When the SPP types the message:

```
ADD $03
```

place the special magnetic tape I/O driver object program (\$0U) in the high-speed paper tape reader. Type Y to copy \$0U on the MOS system file. After the program has been copied, type N to continue system preparation.

6. When the SPP types the message:

**REPLACE FORTRAN**

place the new FORTRAN compiler program on the high-speed paper tape reader. Type Y to copy the FORTRAN compiler to the MOS system file. After the FORTRAN compiler has been copied, type N to continue system preparation.

**Problem 3:**

Prepare an MOS system file for a 620/622 computer system having one 256-track drum memory unit (connected to BIC 020), two standard magnetic tape units with separate controllers, one Teletype unit, one line printer, one high-speed paper tape reader/punch unit, and a 32K core memory. Prepare the system so that only the first 31K of core is used. Make the following logical unit assignments: SF = DR00, PI = PT00, SS = DR01, PO = DR01, BI = PT00, SI = TY00, SO = TY00, LO = LP00, and BO = PT00. Set the default values of \$PGM to 0500, \$LIT to 0377, and \$IAP to 010. Partition the drum memory into six virtual units. Make the first unit 050 sectors longer than the system file, and the next five units 05000, 03000, 01500, 01500, and 01500 sectors long, respectively. Do not list the ISL and do not verify or list the MOS system file.

**Procedure:**

1. Mount the ISL on MT00 (PI).
2. Key in the magnetic tape unit 1 bootstrap and enter the SPP through MT00.
3. Assign logical units for use by the SPP as follows:  
PI = MT00  
PO = DR00  
LO = LP00  
BI = PT00  
SI = TY00
4. Respond to the **BEGIN SYSTEM PREPARATION** message by typing the following control directives on the Teletype (SI):

```
COMP,I,075000,1,0500,0377,010,14
DATE,05/26/70          (optional)
EQUIP,MT00,MT10,TY,TR,TP,PT,LP
EQUIP,DR00(020,050),DR01(020,05000)
EQUIP,DR02(020,03000),DR03(020,01500)
EQUIP,DR04(020,01500),DR05(020,01500)
END,N
```

5. A drum allocation listing of the following form is printed on LO after system preparation is complete:

**DRUM ALLOCATION**

|      |        |        |
|------|--------|--------|
| DR00 | 000012 | 003477 |
| DR01 | 003500 | 010477 |
| DR02 | 010500 | 013477 |
| DR03 | 013500 | 015177 |
| DR04 | 015200 | 016677 |
| DR05 | 016700 | 017777 |

**Problem 4:**

Prepare an MOS system file for a 620/622 computer system having one disc unit connected to BIC 020, one disc unit connected to BIC 022, two 9-track magnetic tape units connected to BIC 024, one Teletype unit, one card reader, one line printer, and 32K core memory. Prepare the system so that only the first 28K of core is used. Make the following logical unit assignments: SF = DK00, PI = CR00, PO = DK01, SS = DK01, BI = MT01, SI = TY00, SO = TY00, LO = LP00, BO = MT00, S1 = DK02, S2 = DK03, S3 = DK40, and S4 = DK41. Partition the first disc unit into four virtual units, with the first virtual unit equal in length to the system file (SF) plus 0100 sectors and the other three virtual units 01500, 01000, and 01000 sectors long, respectively. Partition the second disc unit into five virtual units, each 01200 sectors long. Set the default values of \$PGM to 02000, \$LIT to 01777, and \$IAP to 0500. During system preparation, do not list the ISL, but verify and list the MOS system file.

**Procedure:**

1. Mount the ISL on MT00 (PI).
2. Key in the magnetic tape unit 0 bootstrap and enter the SPP through MT00.
3. Assign logical units for use by the SPP as follows:

```

PI = MT00
PO = DK00
LO = LP00
BI = MT01
SI = TY00

```

4. Respond to the **BEGIN SYSTEM PREPARATION** message by typing the following control directives on the Teletype (SI):

```

COMP,I,067777,3,02000,01777,0500,14
DATE,06/11/70          (optional)
EQUIP,MT00(024),MT01(024),TY,CR,LP

```

## system preparation program

```
EQUIP,DK00(020,0100),DK01(020,01500),DK02(020,01000)
      DK03 (020,01000)
EQUIP,DK40(022,01200),DK41(022,01200),DK42(022,01200)
EQUIP,DK43(022,01200),DK44(022,01200)
ASSIGN,PO=DK01,SS=DK01,BI=MT01,BO=MT00
ASSIGN,S1=DK02,S2=DK03,S3=DK40,S4=DK41
END
```

A disc allocation listing of the following form is printed on LO after system preparation is complete:

### DISC 0 ALLOCATION

|      |        |        |
|------|--------|--------|
| DK01 | 000000 | 003577 |
| DK01 | 003600 | 005277 |
| DK02 | 005300 | 006257 |

### DISC 4 ALLOCATION

|      |        |        |
|------|--------|--------|
| DK40 | 000000 | 001177 |
| DK41 | 001200 | 002377 |
| DK42 | 002400 | 003577 |
| DK43 | 003600 | 004777 |
| DK44 | 005000 | 006257 |

## PROBLEM 5:

Prepare a MOS system file for a 620/622 computer system having one standard magnetic tape unit connected to BIC device address 022, one Teletype unit, one card reader, one high-speed paper tape reader/punch unit, one movable-head disc unit with 9,744 sectors connected to BIC device address 020, one STATOS 21 (620-74) printer/plotter to be used as the system line printer, and 32K core memory. The STATOS 21 printer/plotter programs are not resident on the Installation System Library (ISL) and it is desired to replace the standard line printer routines "RSCB2LPB" (core dump), "\$0Q" (DST), and "LPDP24" (line printer driver) with the STATOS 21 routines "RSCB2LPB", "\$0Q", and "LPST21". Set the default values of \$PGM to 0500, \$LIT to 0377, \$IAP to 010, and number of logical units to twenty. Partition the disc unit into ten virtual units, with the first virtual unit equal in length to the system file (SF), the second virtual unit through the tenth virtual unit into 3000, 800, 500, 900, 700, 100, 300, 200, and 100 sectors long, respectively. Do not list the ISL, but verify the MOS system file.

### Procedure:

### NOTE:

LO cannot be assigned to LP00 unless SPP contains the appropriate line printer driver.

1. Mount the ISL on MT00 (PI).
2. Key in the magnetic tape unit 0 bootstrap and enter the SPP through MT00.
3. Prepare the system preparation control directives on punched cards and place into the input hopper. Make the card reader ready.
4. Assign logical units for use by the SPP as follows:

```

PI = MT00
PO = DK00
LO = TY00
BI = CR00
SI = CR00

```

5. Be prepared to place the appropriate binary-object program replacements into the card reader when the SPP requests the replacements for "RSCB2LPB", "\$0Q", and "LPDP24".
6. Respond to the **Begin System Maintenance** message by making the card reader ready. The following directives, previously prepared, will be read and printed on the Teletype (LO)

```

DATE,07-27-71 (optional)
COMP,I,075777,2,0377,010,20
EQUIP,TY,PT,LP,CR,MOT00(022),DK00(020,0),DK01
(020,3000)
EQUIP,DK02(020,800),DK03(020,500),DK04(020,900),
DK05(020,700)
EQUIP,DK06(020,100),DK07(020,300),DK08(020,200),
DK09(020,100)
REPLACE,$0Q,LPDP24,RSCB2LPB
END,V

```

7. When the SPP types the message:

#### REPLACE RSCB2LPB

place the STATOS 21 core dump routine object program (RSCB2LPB) in the card reader and make ready. Type Y to copy RSCB2LPB onto the MOS system file. After the program has been copied, type N to continue system preparation.

8. When the SPP types the message:

#### REPLACE \$0Q

place the STATOS 21 I/O driver DST object program (\$0Q) in the card reader and make ready. Type Y to copy \$0Q onto the MOS system file. After the program has been copied, type N to continue system preparation.

## **system preparation program**

9. When the SPP types the message:

### **REPLACE LPDP24**

place the STATOS 21 I/O driver object program (LPST21) in the card reader and make ready. Type Y to copy LPST21 onto the MOS system file. After the program has been copied, type N to continue system preparation.

10. Repeat steps 8 and 9 until all required replacements are completed.
11. A disc allocation listing of the following form is printed on L0 (TY00) after system preparation is complete:

### **DISC 0 ALLOCATION**

|      |        |        |
|------|--------|--------|
| DK00 | 000000 | 004544 |
| DK01 | 004545 | 012434 |
| DK02 | 012435 | 014074 |
| DK03 | 014075 | 015060 |
| DK04 | 015061 | 016664 |
| DK05 | 016665 | 020160 |
| DK06 | 020161 | 020324 |
| DK07 | 020325 | 021000 |
| DK08 | 021001 | 021310 |
| DK09 | 021311 | 023017 |

12. After the system has been verified by SPP the following is printed on the L0:

### **MOS SYSTEM READY**

13. The system file is then bootstrapped into the computer via the memory resident MOS and the following is printed on the Teletype (SO):

## SECTION 9 – LANGUAGE PROCESSORS

The basic MOS supports the DAS MR assembler as its language processor. By increasing memory, the FORTRAN IV (ANSI standard) compiler can also be used. The modular design of the MOS allows the inclusion of additional Varian or user language processors through the system preparation procedure.

Both the assembler and compiler exist in stand-alone and MOS configurations. This chapter describes the features of the MOS versions of these language processors.

Status and error messages are given in Section 13.

### DAS MR ASSEMBLER

DAS MR is a two-pass macro assembler that uses the secondary storage device of MOS for the Pass 1 output. It reads a source module from the PI and outputs it on the PO. The Pass 2 source input is input from the SS.

When an END statement is encountered, the SS is repositioned and reread. During Pass 2, the output can be directed to the BO for the object module and the LO for the assembly listing. The SS or PO file, which contains a copy of the source module, can be used as input to a subsequent assembly.

A DAS MR symbol consists of one to six characters, the first of which must be alphabetic, with the rest alphabetic or numeric. Additional alphanumeric characters can be appended to the first six characters of the symbol to form an extended symbol up to the limit imposed by a single line of code. However, only the first six characters are recognized by the assembler.

Assembler language programs (and subroutines) are referred to within MOS by name during system maintenance and system preparation. To name a DAS MR module in MOS, specify from one to eight characters in the title parameter of the /JOB preceding the /ASSEMBLE control directive.

The DAS MR assembler provides for relocatable and absolute object modules. Absolute modules must be explicitly specified with an ORG directive. Otherwise, the modules will be relocatable.



The directives recognized by the DAS MR assembler are:

|      |      |      |      |      |       |
|------|------|------|------|------|-------|
| BES  | DATA | END  | GOTO | MZE  | PZE   |
| BSS  | DETL | ENTR | IFF  | NAME | RETU* |
| CALL | DUP  | EQU  | IFT  | NULL | SET   |
| COMN | EJEC | EXT  | LOC  | OPSY | SPAC  |
| CONT | EMAC | FORM | MAC  | ORG  | SMRY  |

## FORTRAN IV COMPILER

The FORTRAN IV compiler is a one-pass compiler. It inputs a source module from the PI and produces an object module on the BO and an object listing on the LO. No secondary storage is required for a compilation.

When a fatal error is detected (T type, Section 11), the compiler automatically terminates the BO. LO output continues. The compiler reads from the PI file until an END statement is encountered or a control directive is read. Compilation also terminates on detection of an I/O error or an end-of-device, beginning-of-device, or end-of-file indication from I/O control.

FORTRAN IV programs (subroutines, functions, block data, etc.) are referred to within MOS by name during system maintenance and system preparation. To name a FORTRAN IV module in MOS, specify from one to eight characters in the title parameter of the /JOB preceding the /FORTRAN control directive.

The FORTRAN IV compiler output comprises relocatable object modules under all circumstances (e.g., main programs, subprograms, functions, etc.).

The FORTRAN IV compiler has conditional compilation facilities implemented by an X in column 1 of a source statement. When the X appears in the /FORTRAN directive, all source statements with an X in column 1 are compiled with all other statements (viz., the X is treated as a blank). When the X is not present, all conditional statements are ignored by the compiler. X lines are given numbers on the list output in either case, but the source statement is printed only when the X is present.

When performing I/O in FORTRAN, the READ and WRITE statements are used. In these statements, I/O devices are referenced by logical unit numbers. The MOS FORTRAN IV supports logical units 1 through 255 (Figure 3-1). These logical unit numbers are assigned to physical devices (Figure 3-2).

## SECTION 10 — SUPPORT LIBRARY

The MOS system has a comprehensive subroutine library directly available to the user. The library contains mathematical subroutines to support the execution of a FORTRAN IV program, plus many commonly used utility subroutines. To use the library, merely code the proper call in the program, or, for the standard FORTRAN IV functions, implicitly reference the subroutine (e.g.,  $A = \text{SQRT}(B)$  generates a `CALL SQRT(B)`). All calls generate a reference to the required routine, and the loader brings the subroutine into memory and links it to the calling program.

### CALLING SEQUENCE

The subroutines in the support library can be called through a DAS MR or FORTRAN IV program as follows:

#### DAS MR

General form:

```
label CALL S,p(1),p(2),...,p(n)
```

Expansion:

```
label      JMPM      S
           DATA     p(1)
           DATA     p(2)
           .
           .
           .
           DATA     p(n)
```

#### FORTRAN IV

General form:

```
statement number CALL S(p(1),p(2),...,p(n))
```

Generated code:

```
JMPM      S
DATA      p(1)
DATA      p(2)
.
.
.
DATA      p(n)
```

SIXTEEN-BIT NUMBERS (620-SERIES COMPUTERS)

**Single-precision integers** use one 16-bit word. A negative number is in two's complement form. An integer in the range -32,768 to +32,767 can be stored as a single-precision integer.

**Single-precision floating-point numbers** use two consecutive 16-bit words. The exponent (in excess 0200 form) is in bits 14 to 7 of the first word. The mantissa is in bits 6 to 0 of the first word and bits 14 to 0 of the second word. The sign bit of the second word is always zero. A negative number is represented by the one's complement of the first word. Any real number in the range  $10\pm_{38}$  can be stored as a single-precision floating-point number having a precision of six digits.

Single-Precision Floating-Point Number (16-Bit)

|      |    |                        |    |    |    |    |   |   |   |                     |   |   |   |   |   |   |
|------|----|------------------------|----|----|----|----|---|---|---|---------------------|---|---|---|---|---|---|
| Bit  | 15 | 14                     | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6                   | 5 | 4 | 3 | 2 | 1 | 0 |
| n)   | s  | -----Exponent-----     |    |    |    |    |   |   |   | ---High Mantissa--- |   |   |   |   |   |   |
| n+1) | 0  | -----Low Mantissa----- |    |    |    |    |   |   |   |                     |   |   |   |   |   |   |

**Double-precision floating-point numbers** use four consecutive 16-bit words. The exponent (in excess 0200 form) is in bits 7 to 0 of the first word. The mantissa is in the second, third, and fourth words. Bit 17 of the third and fourth words and bits 17 to 8 of the first word are zero. A negative number is represented by the one's complement of the second word. Any real number in the range  $10\pm_{38}$  can be stored as a double-precision floating-point number having a precision of 13 decimal digits.

Double-Precision Floating-Point Numbers (16-bit)

|      |    |                         |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |
|------|----|-------------------------|----|----|----|----|---|---|--------------------|---|---|---|---|---|---|---|
| Bit  | 15 | 14                      | 13 | 12 | 11 | 10 | 9 | 8 | 7                  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| n)   | 0  | 0                       | 0  | 0  | 0  | 0  | 0 | 0 | -----Exponent----- |   |   |   |   |   |   |   |
| N+1) | S  | -----High Mantissa----- |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |
| n+2) | 0  | -----Mid Mantissa-----  |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |
| n+3) | 0  | -----Low Mantissa-----  |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |

## EIGHTEEN-BIT NUMBERS (622-SERIES COMPUTERS)

**Single-precision integers** use one 18-bit word. A negative number is in two's complement form. Any integer in the range -131,072 to +131,071 can be stored as a single-precision integer.

**Single-precision floating-point numbers** use two consecutive 18-bit words. The exponent (in excess 0200 form) is in bits 16 to 9 of the first word. The mantissa is in bits 8 to 0 of the first word, and bits 16 to 0 of the second word. The sign bit of the second word is always zero. A negative number is represented by the one's complement of the first word. Any real number in the range -76,000,000,000 to +76,000,000,000 can be stored as a single-precision floating-point number having a precision of seven digits.

### Single-Precision Floating-Point Number (18-Bit)

|      |    |                        |    |    |    |    |    |    |   |                         |   |   |   |   |   |   |   |   |
|------|----|------------------------|----|----|----|----|----|----|---|-------------------------|---|---|---|---|---|---|---|---|
| Bit  | 17 | 16                     | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8                       | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| n)   | s  | -----Exponent-----     |    |    |    |    |    |    |   | -----High Mantissa----- |   |   |   |   |   |   |   |   |
| n+1) | 0  | -----Low Mantissa----- |    |    |    |    |    |    |   |                         |   |   |   |   |   |   |   |   |

**Double-precision floating-point numbers** use four consecutive 18-bit words. The exponent (in excess 0200 form) is in bits 7 to 0 of the first word. The mantissa is in the second, third, and fourth words. Bit 17 of the third and fourth words and bits 17 to 8 of the first word are zero. A negative number is represented by the one's complement of the second word. Any real number in the range -76,000,000,000 to +76,000,000,000 can be stored as a double-precision floating-point number having a precision of 15 decimal digits.

### Double-Precision Floating-Point Number (18-Bit)

|      |    |                         |    |    |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |
|------|----|-------------------------|----|----|----|----|----|----|---|---|--------------------|---|---|---|---|---|---|---|
| Bit  | 17 | 16                      | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7                  | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| n)   | 0  | 0                       | 0  | 0  | 0  | 0  | 0  | 0  | 0 | 0 | -----Exponent----- |   |   |   |   |   |   |   |
| n+1) | s  | -----High Mantissa----- |    |    |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |
| n+2) | 0  | -----Mid Mantissa-----  |    |    |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |
| n+3) | 0  | -----Low Mantissa-----  |    |    |    |    |    |    |   |   |                    |   |   |   |   |   |   |   |

# SUBROUTINE DESCRIPTIONS

The following definitions and notation apply to the subroutine descriptions given in this section:

| Notation | Meaning                                                                                                                                                |
|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| AB       | Hardware A and B registers                                                                                                                             |
| AC       | Four-word software accumulator for double-precision and real numbers.                                                                                  |
| ACCZ     | Four-word software accumulator for complex numbers (defined as labelled COMMON block \$IMAG and where the results of all complex functions are placed) |
| d        | A double-precision number                                                                                                                              |
| f        | Two-word fixed-point number                                                                                                                            |
| i        | An integer                                                                                                                                             |
| j        | A double-precision integer                                                                                                                             |
| r        | A real number                                                                                                                                          |
| X        | Hardware X register                                                                                                                                    |
| z        | A complex number                                                                                                                                       |
| **       | Exponentiation                                                                                                                                         |
| /        | Division                                                                                                                                               |
| \$       | A character commonly used as the first character of a subroutine name                                                                                  |

The external references in Figure 10-1 refer to items in both Figures 10-1 and 10-2. Similarly, the subroutines called in Figure 10-2 refer to items in both Figures 10-1 and 10-2. When a subroutine has more than one name, it is indicated by multiple calls under Calling Sequence.

| Name   | Function                                                                                                                            | Calling Sequence   | External References                                                             |
|--------|-------------------------------------------------------------------------------------------------------------------------------------|--------------------|---------------------------------------------------------------------------------|
| \$HE   | In A, compute $i1**i2$                                                                                                              | CALL \$HE,i2       | \$QS, \$SE, \$PE,<br>\$HS, \$QK                                                 |
| \$PE   | In AB, compute $r**i$                                                                                                               | CALL \$PE,i        | \$SE, \$QS, \$QE                                                                |
| \$QE   | In AB, compute $r1**r2$                                                                                                             | CALL \$QE,r2       | ALOG, \$QM, EXP,<br>\$SE                                                        |
| ALOG   | In AB, compute $\ln r$ . If negative, error exit with $A = B = 0$ and overflow = 1. If zero, set result to maximum negative number. | CALL ALOG,r        | \$ER, \$QS, \$QK,<br>\$QM, XDMU, XDAD,<br>\$FMS, \$NML, \$XDDI,<br>\$XDSU, \$SE |
| EXP    | In AB, compute $e**r$                                                                                                               | CALL EXP,r         | XDMU, \$QK, \$QL,<br>\$QM, \$QN, \$SE                                           |
| ATAN   | In AB, compute $\arctan r$                                                                                                          | CALL ATAN,r        | \$QM, \$QL, \$QN,<br>\$QK, \$SE                                                 |
| COSINE | In AB, compute $\cos r$                                                                                                             | CALL COS,r         | SIN, \$QL, \$SE                                                                 |
| SINE   | In AB, compute $\sin r$                                                                                                             | CALL SIN,r         | \$QM, XDMU, XDAD,<br>\$NML, \$FMS, \$SE                                         |
| \$SE   | To transfer a list of N parameters from a calling program to a called subprogram                                                    | CALL \$SE, N, LIST | None                                                                            |

Figure 10-1. DAS Coded Subroutines (1 of 6)

| Name    | Function                                                                                                                   | Calling Sequence                   | External References      |
|---------|----------------------------------------------------------------------------------------------------------------------------|------------------------------------|--------------------------|
| SQRT    | In AB, compute square root of $r$                                                                                          | CALL SQRT, $r$                     | XDDI, \$FSM, \$SE, XDIV  |
| FMULDIV | In AB, compute $r1*r2$ with \$QM, or $r1/r2$ with \$QN. If result overflows, error exit with $A = B = 0$ and overflow = 1. | CALL \$QM, $r2$<br>CALL \$QN, $r2$ | XDMU, \$FMS, XCCI, \$SE  |
| FADDSUB | In AB, compute $r1 + r2$ with \$QK, or $r1 - r2$ with \$QL                                                                 | CALL \$QK, $r2$<br>CALL \$QL, $r2$ | \$SE, \$FSM, \$NML, \$ER |
| SEPMAN  | Separate mantissa and characteristic of $r$ into AB and X respectively                                                     | CALL \$FMS<br>CALL \$FSM           | None                     |
| FNORMAL | In AB, normalize $r$                                                                                                       | CALL \$NML                         | XDCO                     |
| XDDIV   | In AB, compute $f1/f2$                                                                                                     | CALL XDDI, $f2$                    | XDSU, XDCO, XDIV, XMUL   |
| XDMULT  | In AB, compute $f1*f2$                                                                                                     | CALL XDMU, $f2$                    | XDAD, XDCO, XMUL         |
| XDADD   | In AB, compute $f1 + f2$                                                                                                   | CALL XDAD, $f2$                    | None                     |
| XDSUB   | In AB, compute $f1 - f2$                                                                                                   | CALL XDSU, $f2$                    | None                     |
| XDCOMP  | In AB, compute negative of $f$                                                                                             | CALL XDCO                          | None                     |

Figure 10-1. DAS Coded Subroutines (2 of 6)

| Name    | Function                                                                     | Calling Sequence         | External References |
|---------|------------------------------------------------------------------------------|--------------------------|---------------------|
| \$FLOAT | In AB, convert the i in A to floating-point and, for \$QS, store result in r | CALL \$PC<br>CALL \$QS,r | \$SE                |
| \$IFIX  | In A, convert the r in AB to i and, for \$HS, store result in i              | CALL \$IC<br>CALL \$HS,i | \$SE                |
| IABS    | In A, compute absolute i                                                     | CALL IABS,i              | \$SE                |
| ABS     | In AB, compute absolute r                                                    | CALL ABS,r               | \$SE                |
| ISIGN   | Set the sign of i1, in A, equal to that of i2                                | CALL ISIGN,i2            | \$SE                |
| SIGN    | Set the sign of r1, in AB, equal to that of r2                               | CALL SIGN,r2             | \$SE                |
| \$HN    | In A, compute i1/i2                                                          | CALL \$HN,i2             | \$SE, XDIV          |
| \$HM    | In A, compute i1*i2                                                          | CALL \$HM,i2             | \$SE, XMUL          |
| XMUL    | Software emulation of hardware multiplication                                | CALL XMUL,i              | None                |
| XDIV    | Software emulation of hardware division                                      | CALL XDIV,i              | None                |

Figure 10-1. DAS Coded Subroutines (3 of 6)



|         |                                                                                                                            |                                                                |                                                                       |
|---------|----------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|-----------------------------------------------------------------------|
| DSINCOS | In AC, compute $\sin d$ or $\cos d$                                                                                        | CALL \$DSI,d<br>CALL \$DSIN,d<br>CALL \$DCO,d<br>CALL \$DCOS,d | \$STO, \$DMP, \$DIT,<br>\$DFR, CHEB, \$SE,<br>\$DLO                   |
| DATAN   | In AC, compute $\arctan d$                                                                                                 | CALL \$DAT,d<br>CALL \$DATAN,d                                 | \$DLO, \$STO, \$DAD,<br>\$DSU, IF, \$SE,<br>AC, \$DMP, \$DDI,<br>POLY |
| DEXP    | In AC, compute exponential d                                                                                               | CALL \$DEX,d<br>CALL \$EXP,d<br>CALL \$TWOX,d                  | \$DLO, \$STO, \$DMP,<br>\$DDI, \$SE, AC,<br>\$DIT, CHEB, IF,<br>\$DFR |
| DLOG    | In AC, compute $\ln d$                                                                                                     | CALL \$DLOG,d<br>CALL \$DLN,d                                  | \$DLO, \$STO, \$DAD,<br>POLY, IF, \$SE,<br>AC, \$DSU, \$DMP,<br>\$DDI |
| POLY    | In AC, compute double-precision polynomial with t terms, coefficient list starting at address c, and argument at address y | CALL POLY,t,c,y                                                | \$DLO, \$DAD, \$DMP                                                   |
| CHEB    | In AC, compute shifted Chebyshev polynomial series with t + 1 terms and coefficient list starting at address c             | CALL CHEB,t,c                                                  | \$DLO, \$STO, \$DAD,<br>\$DSU, \$DMP                                  |

Figure 10-1. DAS Coded Subroutines (4 of 6)

|          |                                                              |                                                                |                                                          |
|----------|--------------------------------------------------------------|----------------------------------------------------------------|----------------------------------------------------------|
| DSQRT    | In AC, compute square root of d                              | CALL \$DSQ,d<br>CALL \$DSQR,d                                  | \$DLO, \$STO, \$DNO,<br>\$DAD, \$DMP, \$DDI,<br>\$SE, AC |
| \$DFR    | In AC, compute fractional part of d                          | CALL \$DFR,d                                                   | \$DLO, \$DNO, \$DSU,<br>\$DIT, AC, \$SE                  |
| IDINT    | In AC, compute integral part of d                            | CALL \$DIT,d<br>CALL \$IDIN,d                                  | \$DNO, \$SE                                              |
| DMULT    | In AC, compute $d1 * d2$                                     | CALL \$DMP,d2<br>CALL \$ZM,d2                                  | \$DLO, \$STO, \$DNO,<br>\$DAD, AC, \$SE                  |
| DIVIDE   | In AC, compute $d1 / d2$                                     | CALL \$DDI,d2<br>CALL \$ZN,d2                                  | \$DLO, \$STO, \$DNO,<br>\$DSU, AC, \$SE                  |
| DADDSUB  | In AC, compute $d1 + d2$ with \$DAD, or $d1 - d2$ with \$DSU | CALL \$DAD,d2<br>CALL \$DSU,d2<br>CALL \$ZK,d2<br>CALL \$ZL,d2 | \$STO, \$DLO, \$DNO,<br>AC                               |
| DNORMAL  | In AC, normalize d                                           | CALL \$DNO                                                     | \$SE                                                     |
| DLOADAC  | Load AC with d                                               | CALL \$DLO,d<br>CALL \$ZF,d                                    | AC, \$SE                                                 |
| DSTOREAC | Store AC in d                                                | CALL \$STO,d<br>CALL \$ZS,d                                    | AC, \$SE                                                 |
| RLOADAC  | Load A with double-precision mantissa sign word from AC      | CALL \$ZI                                                      | AC                                                       |

Figure 10-1. DAS Coded Subroutines (5 of 6)

|                  |                                                                                                                                         |                 |       |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------|-----------------|-------|
| SINGLE           | In AB, convert the d in AC to r                                                                                                         | CALL \$RC       | AC    |
| DOUBLE           | In AC, convert the r in AB to d                                                                                                         | CALL \$YC       | AC    |
| DBLECOMP         | In AC, compute negative of the d in AC                                                                                                  | CALL \$ZC       | AC    |
| \$3S             | Store AB in memory address m                                                                                                            | CALL \$3S,m     | \$SE  |
| SNAP             | Print the contents of core on logical unit (LO)<br>(A) = starting address<br>(B) = ending address                                       | CALL SNAP       | IOCS  |
| A2MT<br>620 only | Translate in memory a character string of length n starting at s and ending at e from eight-bit ASCII to six-bit magnetic tape BCD code | CALL A2MT,n,s,e | None  |
| MT2A<br>620 only | Translate in memory a character string of length n starting at s and ending at e from six-bit magnetic tape BCD code to eight-bit ASCII | CALL MT2A,n,s,e | None  |
| DEBUG            | Provide an execution address for the DEBUG package                                                                                      | CALL DEBUG      | DBG\$ |

Figure 10-1. DAS Coded Subroutines (6 of 6)

| Name | Function                         | Calling Sequence | External Ref.                                           |
|------|----------------------------------|------------------|---------------------------------------------------------|
| \$9E | Compute $\text{ACCZ}^{**i}$      | CALL \$9E(i)     | \$SE, IABS, \$8F,<br>\$8M, \$8N                         |
| CCOS | In ACCZ, compute $\cos z$        | CALL CCOS(z)     | \$SE, CSIN, \$8F,<br>\$8K, \$8S                         |
| CSIN | In ACCZ, compute $\sin z$        | CALL CSIN(z)     | \$SE, EXP, \$QN,<br>SIN, \$QK, \$QM,<br>COS, \$QL, \$8F |
| CLOG | In ACCZ, compute $\ln z$         | CALL CLOG(z)     | \$SE, ALOG, \$QM,<br>\$QK, \$QN, ATAN2,<br>\$8F         |
| CEXP | In ACCZ, compute exponential $z$ | CALL CEXP(z)     | \$SE, EXP, COS,<br>SQM, SIN, \$8F                       |

Figure 10-2. FORTRAN IV Coded Subroutines (1 of 6)

| Name  | Function                                                              | Calling Sequence  | External Ref.                            |
|-------|-----------------------------------------------------------------------|-------------------|------------------------------------------|
| CSQRT | In ACCZ, compute square root of z                                     | CALL CSQRT(z)     | \$SE, SQRT, CABS, \$QK, \$QN, \$8F       |
| CABS  | In AB, compute absolute z                                             | CALL CABS(z)      | \$SE, SQRT, \$QM, \$QK                   |
| CONJG | In ACCZ, compute conjugate of z                                       | CALL CONJG(z)     | \$SE, \$8F                               |
| \$AK  | Add r to real part of ACCZ                                            | CALL \$AK(r)      | \$SE, \$8S, \$QK, \$8F                   |
| \$AL  | Subtract r from the real part of ACCZ                                 | CALL \$AL(r)      | \$SE, \$8S, \$QL, \$8F                   |
| \$AM  | Multiply ACCZ by r                                                    | CALL \$AM(r)      | \$SE, \$8S, \$QM, \$8F                   |
| \$AN  | Divide ACCZ by r                                                      | CALL \$AN(r)      | \$SE, \$8S, \$QM, \$8F                   |
| \$AC  | Convert AC to z and store in ACCZ                                     | CALL \$AC         | \$3S, CMLPX                              |
| CMLPX | Load ACCZ with a value having a real part r1 and an imaginary part r2 | CALL CMLPX(r1,r2) | \$SE, \$8F                               |
| \$8K  | Add z to ACCZ                                                         | CALL \$8K(z)      | \$SE, \$8S, \$QK, \$8F                   |
| \$8L  | Subtract z from ACCZ                                                  | CALL \$8L(z)      | \$SE, \$8S, \$QL, \$8F                   |
| \$8M  | Multiply ACCZ by z                                                    | CALL \$8M(z)      | \$SE, \$8S, \$QM, \$QL, \$QK, \$8F       |
| \$8N  | Divide ACCZ by z                                                      | CALL \$8N(z)      | \$SE, \$8S, \$QM, \$QK, \$QN, \$QL, \$8F |

Figure 10-2. FORTRAN IV Coded Subroutines (2 of 6)

| Name   | Function                             | Calling Sequence   | External Ref.                                         |
|--------|--------------------------------------|--------------------|-------------------------------------------------------|
| \$ZD   | Compute negative of z                | CALL \$ZD          | \$8S, \$8F                                            |
| AIMAG  | Load AB with the imaginary part of z | CALL AIMAG(z)      | \$SE                                                  |
| \$OC   | Load AB with the real part of ACCZ   | CALL \$OC          | \$8S                                                  |
| REAL   | Load AB with the real part of z      | CALL REAL(z)       | \$SE                                                  |
| \$8F   | Load ACCZ with z                     | CALL \$8F(z)       | \$SE                                                  |
| \$8S   | Store ACCZ in z                      | CALL \$8S(z)       | \$SE, \$3S                                            |
| \$XE   | Compute $d**i$ where d is in AC      | CALL \$XE(i)       | \$SE, \$ZF, MOD, \$ZM, \$HN, \$ZN                     |
| \$YE   | Compute $d**r$ where d is in AC      | CALL \$YE(r)       | \$SE, \$ZS, DBLE, \$ZE, \$ZF                          |
| \$ZE   | Compute $d1**d2$ where d1 is in AC   | CALL \$ZE(d2)      | \$SE, \$ZS, DEXP, DLOG, \$ZM                          |
| DATAN2 | In AC, compute arctan ( $d1/d2$ )    | CALL DATAN2(d1,d2) | \$SE, \$ZF, \$ZS, \$ZI, \$ER, \$ZN, \$ZL, \$ZK, DATAN |
| DLOG10 | In AC, compute log d                 | CALL DLOG10(d)     | \$SE, DLOG, \$ZM                                      |
| DMOD   | In AC, compute d1 modulo d2          | CALL DMOD(d1,d2)   | \$SE, DINT, \$ZF, \$ZN, \$ZS, \$ZM, \$ZL, \$ZC        |
| DINT   | In AC, compute integer portion of d  | CALL DINT(d)       | \$SE, \$ZF, \$JC, \$XC                                |
| DABS   | In AC, compute absolute d            | CALL DABS(d)       | \$SE, \$ZF, \$ZI,                                     |

Figure 10-2. FORTRAN IV Coded Subroutines (3 of 6)

| Name   | Function                                                 | Calling Sequence           | External Ref.<br>\$ZN              |
|--------|----------------------------------------------------------|----------------------------|------------------------------------|
| DMAX1  | In AC, select the maximum value in the set d1, d2,...,dn | CALL DMAX1(d1,d2,...,dn,0) | \$SE, \$ZF, \$ZS, \$FA, \$ZL, \$ZI |
| DMIN1  | In AC, select the minimum value in the set d1, d2,...,dn | CALL DMIN1(d1,d2,...,dn,0) | \$SE, \$ZF, \$ZS, \$FA, \$ZL, \$ZI |
| DSIGN  | Set the sign of d1 equal to that of d2                   | CALL DSIGN (d1,d2)         | \$SE, \$ZF, \$ZI, \$ZN             |
| \$YK   | Add r to AC                                              | CALL \$YK(r)               | \$SE, \$ZS, DBLE, \$ZK             |
| \$YL   | Subtract r from AC                                       | CALL \$YL(r)               | \$SE, \$ZS, DBLE, \$ZL, \$ZC       |
| \$YM   | Multiply AC by r                                         | CALL \$YM(r)               | \$SE, \$ZS, DBLE, \$ZM             |
| \$YN   | Divide AC by r                                           | CALL \$YN(r)               | \$SE, \$ZS, DBLE, \$ZF, \$ZN       |
| DBLE   | In AC, convert r to d                                    | CALL DBLE(r)               | \$SE, \$YC                         |
| \$XC   | In AC, convert i to d where i is in A                    | CALL \$XC                  | \$PC, \$YC                         |
| TANH   | In AB, compute tanh r                                    | CALL TANH(r)               | \$SE, \$QK, EXP, \$QL, \$QN        |
| ATAN2  | In AB, compute arctan (r1/r2)                            | CALL ATAN2(r1,r2)          | \$SE, \$ER, ATAN, \$QK, \$QL, \$QN |
| ALOG10 | In AB, compute log r                                     | CALL ALOG10(r)             | \$SE, ALOG, \$QM                   |
| AMOD   | In AB, compute r1 modulo r2                              | CALL AMOD(r1,r2)           | \$SE, AINT, \$QN,                  |

Figure 10-2. FORTRAN IV Coded Subroutines (4 of 6)

| Name  | Function                                                                 | Calling Sequence           | External Ref.<br>\$QM, \$QL |
|-------|--------------------------------------------------------------------------|----------------------------|-----------------------------|
| AINT  | In AB, truncate r                                                        | CALL AINT(r)               | \$SE, \$IC, \$PC            |
| AMAX1 | In AB, select the maximum value in the set r1,r2,...,rn                  | CALL AMAX1(r1,r2,...,rn,0) | \$SE, \$FA, \$QL            |
| AMIN1 | In AB, select the minimum value in the set r1,r2,...,rn                  | CALL AMIN1(r1,r2,...,rn,0) | \$SE, \$FA, \$QL            |
| AMAX0 | In AB, select the maximum value in the set i1,i2,...,in and convert to r | CALL AMAX0(i1,i2,...,in,0) | \$SE, \$FA, FLOAT           |
| AMIN0 | In AB, select the minimum value in the set i1,i2,...,in and convert to r | CALL AMIN0(i1,i2,...,in,0) | \$SE, \$FA, FLOAT           |
| DIM   | In AB, compute the positive difference between r1 and r2                 | CALL DIM(r1,r2)            | \$SE, \$QL                  |
| FLOAT | In AB, convert i to r                                                    | CALL FLOAT(i)              | \$SE, \$PC                  |
| SNGL  | In AB, convert d to r                                                    | CALL SNGL(d)               | \$SE, \$ZF, \$RC            |
| MAX0  | In A, select the maximum value in the set i1,i2,...,in                   | CALL MAX0(i1,i2,...,in,0)  | \$SE, \$FA                  |
| MIN0  | In A, select the minimum value in the set i1,i2,...,in                   | CALL MIN0(i1,i2,...,in,0)  | \$SE, \$FA                  |
| MAX1  | In A, select the maximum value in the set r1,r2,...,rn and convert to i  | CALL MAX1(r1,r2,...,rn,0)  | \$SE, \$FA, \$QL, IFIX      |
| MIN1  | In A, select the minimum                                                 | CALL MIN1(r1,r2,           | \$SE, \$FA, \$QL.           |

Figure 10-2. FORTRAN IV Coded Subroutines (5 of 6)



| Name | Function                                                      | Calling Sequence | External Ref.    |
|------|---------------------------------------------------------------|------------------|------------------|
|      | value in the set r1,r2,<br>...,rn and convert to i            | ...,rn,0)        | IFIX             |
| MOD  | In A, compute i1 modulo i2                                    | CALL MOD(i1,i2)  | \$SE, \$HN, \$HM |
| INT  | In A, truncate r and<br>convert to i                          | CALL INT(r)      | \$SE, \$IC       |
| IDIM | In A, compute the positive<br>difference between i1 and<br>i2 | CALL IDIM(i1,i2) | \$SE             |
| IFIX | In A, convert r to i                                          | CALL IFIX(r)     | \$SE, \$IC       |
| \$JC | In AC, convert d to i and<br>store result in A                | CALL \$JC        | \$RC, \$IC       |

Figure 10-2. FORTRAN IV Coded Subroutines (6 of 6)

## SECTION 11 — MOS OPERATING PROCEDURES

The installation system library (ISL) of the MOS is available on punched paper tape, or, on systems having 12K of memory, on magnetic tape or punched cards. From the ISL, the user performs a system preparation to obtain a system file (Section 8). The following procedures assume the presence of a system file and deal only with bootstrapping and initialization of MOS.

### DEVICE INITIALIZATION

#### CARD READER

- a. Turn on the card reader.
- b. Place two blank cards after the last control-directive card of the input deck.
- c. Place the input deck in the card hopper.
- d. Press CLEAR and START.

#### CARD PUNCH

- a. Turn on the card punch.
- b. Place blank cards in the card hopper.
- c. Press START.

#### 33/35 ASR TELETYPE

- a. Turn on the Teletype.
- b. Set the Teletype in off-line mode and simultaneously press the CONTROL and D, then the CONTROL and T, and finally CONTROL and Q keys.
- c. Set the Teletype on-line.

## **HIGH-SPEED PAPER TAPE READER**

- a. Turn on the paper tape reader.
- b. Position the input paper tape in the reader with blank leader at the reading station and close the reading gate.
- c. Set the LOAD/RUN switch to RUN.

## **MAGNETIC TAPE UNIT**

- a. Turn on the magnetic tape unit.
- b. Mount the input magnetic tape.
- c. Position the magnetic tape to the loading point.
- d. Ready the magnetic tape unit so it can be used by the computer.

## **MAGNETIC DRUM UNIT** (*Models 620-46 through 620-49*)

- a. Turn on the drum unit.
- b. Wait for the drum unit to reach operating speed.

## **FIXED-HEAD DISC UNIT** (*Model 620-38C*)

- a. Press the AC POWER switch.
- b. Wait for the AC POWER indicator to light.
- c. Press the DC POWER switch.

## **MOVING-HEAD DISC UNIT** (*Model 620-39*)

- a. Turn on the disc unit.
- b. Set the START/STOP switch to START.
- c. Wait for the disc unit to reach operating speed (READY indicator lights).
- d. Turn off WRITE PROTECT.

## **MOVING-HEAD DISC UNIT** (*Models 620-40, -41*)

- a. Turn on the disc unit.
- b. Wait for the disc unit to reach operating speed (DISC READY indicator lights).

## OOTSTRAP

Enter the bootstrap loading routine for the device containing the system file in memory as follows (Figure 11-1):

- a. Press REPEAT (and position STEP/RUN to STEP for 620/f).
- b. Enter instruction 054000 (store A relative to P) in the instruction register.
- c. Reset the P register.
- d. Enter a bootstrap instruction in the A register.
- e. Press STEP (START for 620/f).
- f. Reset the A register.
- g. Repeat steps d, e, and f for each bootstrap instruction.

To initiate the bootstrap, reset the A, B, X, P, and U (I for 620/f) registers. Then, press SYSTEM RESET and RUN (for 620/f press RESET, position STEP/RUN to RUN, and press START).

| Address | Magnetic<br>Tape<br>Unit | 620-46<br>to -49<br>Drum | 620-38C<br>Fixed-<br>Head<br>Disc | 620-39<br>Moving-<br>Head<br>Disc | 620-40, -41<br>Moving-<br>Head<br>Disc | 620-37<br>Moving-<br>Head<br>Disc |
|---------|--------------------------|--------------------------|-----------------------------------|-----------------------------------|----------------------------------------|-----------------------------------|
| 00000   | 104110                   | 1000yy                   | 1000yy                            | 005003                            | 005006                                 | 005003                            |
| 00001   | 101210                   | 006020                   | 006020                            | 101316                            | 010024                                 | 100416                            |
| 00002   | 000005                   | 000012                   | 000012                            | 000013                            | 140034                                 | 100216                            |
| 00003   | 001000                   | 010014                   | 010014                            | 100716                            | 001002                                 | 103116                            |
| 00004   | 000001                   | 1031xx                   | 1031xx                            | 100216                            | 001001                                 | 101016                            |
| 00005   | 030016                   | 006120                   | 006120                            | 103016                            | 120034                                 | 000010                            |
| 00006   | 100010                   | 000350                   | 000350                            | 000035                            | 100015                                 | 001000                            |
| 00007   | 102510                   | 1031yy                   | 1031yy                            | 101516                            | 1000yy                                 | 000004                            |
| 00010   | 055000                   | 1000xx                   | 1000xx                            | 000025                            | 1031xx                                 | 100416                            |
| 00011   | 005144                   | 100014                   | 100014                            | 001000                            | 006120                                 | 100316                            |
| 00012   | 101110                   | 103214                   | 103214                            | 000007                            | 000121                                 | 103116                            |
| 00013   | 000007                   | 1010xx                   | 1010xx                            | 010033                            | 1031yy                                 | 100021                            |
| 00014   | 101210                   | 001001                   | 001001                            | 100716                            | 1000xx                                 | 014011                            |
| 00015   | 001001                   | 001000                   | 001000                            | 100416                            | 103215                                 | 103120                            |
| 00016   | 001000                   | 000013                   | 000013                            | 103216                            | 001040                                 | 124011                            |
| 00017   | 000012                   |                          |                                   | 1000yy                            | 000026                                 | 103121                            |
| 00020   |                          |                          |                                   | 1031xx                            | 100415                                 | 100020                            |
| 00021   |                          |                          |                                   | 120036                            | 005122                                 | 100016                            |
| 00022   |                          |                          |                                   | 1031yy                            | 101415                                 | 101416                            |
| 00023   |                          |                          |                                   | 1000xx                            | 000002                                 | 000022                            |
| 00024   |                          |                          |                                   | 100016                            | 001000                                 | 101516                            |
| 00025   |                          |                          |                                   | 101216                            | 000022                                 | 000000                            |
| 00026   |                          |                          |                                   | 000025                            | 100115                                 | 001000                            |
| 00027   |                          |                          |                                   | 101016                            | 101015                                 | 001001                            |
| 00030   |                          |                          |                                   | 000027                            | 000001                                 | 000550                            |
| 00031   |                          |                          |                                   | 001010                            | 005144                                 | 000000                            |
| 00032   |                          |                          |                                   | 000000                            | 001000                                 |                                   |
| 00033   |                          |                          |                                   | 001000                            | 000027                                 |                                   |
| 00034   |                          |                          |                                   | 001001                            | 001520                                 |                                   |
| 00035   |                          |                          |                                   | 000312                            |                                        |                                   |
| 00036   |                          |                          |                                   | 000550                            |                                        |                                   |

Where xx = even BIC address and yy = odd BIC address.

**Figure 11-1. MOS System File Bootstrap Routines**

## SYSTEM (RE)INITIALIZATION

The executive component of MOS (re)initializes the system:

- a. At bootstrap time
- b. When /JOB is read
- c. When /ENDJOB is read
- d. When the operator resets the A, B, X, P, and U (I for the 620/f) registers, presses the SYSTEM RESET, and presses RUN (for the 620/f, the operator presses RESET, positions STEP/RUN to RUN, and presses START).

Upon (re)initialization:

- a. All logical unit assignments are set to their default values.
- b. System flags denoting errors, processor options, and loader options are reset.

To (re)execute the user's program in memory:

- a. Set the P register to 000002 and the instruction register to zero.
- b. Press SYSTEM RESET (RESET for 620/f).
- c. Press RUN (for 620/f, position STEP/RUN to RUN and press START).

To enter the dump program prior to (re)initialization through the executive:

- a. Set the P register to 000004 and the instruction register to zero.
- b. Press SYSTEM RESET (RESET for 620/f).
- c. Press RUN (for 620/f, position STEP/RUN to RUN and press START).



## **SECTION 12 — USER-CODED I/O DRIVERS**

MOS permits augmenting the system with I/O drivers coded by the user. To develop an I/O routine and place it in MOS:

- a. Code the driver according to the applicable I/O device specification.
- b. Assemble the driver using the DAS MR assembler.
- c. Add the driver to the system file using the system preparation program.
- d. Use the driver through I/O control calls.



# DEVICE SPECIFICATION TABLE

When a driver controls a single logical unit, the first 16 or more words comprise the device specification table (DST). When multiple units are controlled, such as magnetic tape units, a DST is required for each. Each DST is a separate assembly linked to its driver through externals. The DST:

- a. Transfers parameters of the user I/O request to the driver
- b. Determines if an I/O driver can accommodate an I/O request
- c. Obtains the results of I/O function requests

Figure 12-1 lists I/O driver entry names and their corresponding device names and device addresses as recognized by MOS Input/Output Control. The device addresses are omitted for those peripheral devices which are presently not supported by VDM. If a user replaces an existing driver or incorporates a driver for one of the devices which is not supported, it is suggested that the listed entry names, device names, and device addresses be retained.

DST words have the following significance:

| Word | Meaning                              |
|------|--------------------------------------|
| 0    | Actual number of transfers           |
| 1    | I/O status                           |
| 2-3  | I/O driver name                      |
| 4    | I/O request flag and operation code  |
| 5    | Count parameter                      |
| 6    | Address parameter                    |
| 7    | Address of I/O checking routine      |
| 8    | Address of I/O checking routine      |
| 9    | Address of reading routine           |
| 10   | Address of writing routine           |
| 11   | Address of write-end-of-file routine |
| 12   | Address of rewind routine            |
| 13   | Address of skip-files routine        |
| 14   | Address of skip-records routine      |
| 15   | Address of function routine          |
| 16-n | Defined by driver                    |

**WORD 0**

Modified by: I/O driver  
 Used by: User programs when information on the number of transfers is required

Word 0 is associated with the entry name of the I/O driver since all remaining DST words are referenced relative to word 0. I/O driver entry names comprise three characters, the first of which is the dollar sign (\$); the second, a number (0 through 9); and the third, a letter or number.

Word 0 contains the number of words transferred for an I/O reading or writing request, or the number of files or records remaining to be skipped for a skip request. The I/O driver puts this information into word 0 of the DST.

**WORD 1**

Modified by: I/O driver  
 Used by: I/O control and user programs

Bits 0 through 2 of word 1 are examined by I/O control during a status request call to determine the appropriate return as follows:

| Value | Meaning                            |
|-------|------------------------------------|
| 0     | Normal return                      |
| 1     | Error                              |
| 2     | End of file                        |
| 3     | End or beginning of device         |
| 4     | Last operation not complete (busy) |

Bits 3 through 15 (17 on 18-bit computers) are not examined by I/O control. However, for uniformity in the I/O driver, the following meanings are ascribed to the bit positions:

| Bit | Meaning                          |
|-----|----------------------------------|
| 3-5 | Temporary storage for I/O driver |
| 6   | Last operation was rewind        |
| 7   | Odd-length record detected       |
| 8   | Error detected                   |

| <b>Bit</b> | <b>Meaning</b>                           |
|------------|------------------------------------------|
| 9          | Unit ready                               |
| 10         | Unit rewinding                           |
| 11         | End of file detected                     |
| 12         | End of device detected                   |
| 13         | Beginning of device detected             |
| 14         | Last operation ignored                   |
| 15         | Status not valid                         |
| 16         | Not used (18-bit computers only)         |
| 17         | Status not valid (18-bit computers only) |

## **WORDS 2 and 3**

|              |                               |
|--------------|-------------------------------|
| Modified by: | Not modified                  |
| Used by:     | I/O control and user programs |

Words 2 and 3 contain the four-character ASCII name of the peripheral device for the I/O driver (Figure 3-2).

## **WORD 4**

|              |                            |
|--------------|----------------------------|
| Modified by: | I/O control and I/O driver |
| Used by:     | I/O control and I/O driver |

When word 4 is positive, I/O control is advised that an I/O request is in process or pending and a new I/O request must wait. When word 4 is negative, I/O control is notified that the I/O driver is available. When an I/O request is made for an available I/O driver, I/O control puts the function code from the user's I/O request in bits 7 through 0 of word 4, making it positive.

## **WORD 5**

|              |                            |
|--------------|----------------------------|
| Modified by: | I/O control and I/O driver |
| Used by:     | I/O driver                 |

When an I/O request is made for an I/O driver and no other request is pending (word 4 negative), I/O control puts the count (if greater than zero) from the user's I/O call in word 5.

## **WORD 6**

|              |                            |
|--------------|----------------------------|
| Modified by: | I/O control and I/O driver |
| Used by:     | I/O driver                 |

When an I/O request is made for an I/O driver and no other request is pending (word 4 negative), I/O control puts the data address from the user's I/O call in word 6. The most significant bit is always zero.

**WORDS 7 AND 8**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

Words 7 and 8, which contain the same address, are used as follows:

- a. Before I/O control passes to the I/O driver to perform the requested I/O function, it transfers to the I/O driver by a Jump and Mark Indirect (JMPM\*) instruction to the address of word 7. Then, the I/O driver determines if the peripheral device is available. If so, it sets a positive condition code in the A register, permitting I/O control to enter the I/O driver a second time to process the request. If the device is unavailable, the A register is set to zero, signaling I/O control not to make an I/O request at this time.
- b. Every time a call requesting status is made to I/O control, it checks the I/O driver request flag (most significant bit of word 4). If no request is in process, control passes to the I/O driver with a JMPM\* to the address given in word 7. The I/O driver sets the condition code in the A register before returning control to I/O control.

**WORD 9**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can read, word 9 contains the address used by I/O control when reading is requested of this I/O driver. If the I/O driver cannot read, word 9 is zero.

To read, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the reading is complete, the I/O driver returns control to I/O control with a Jump Indirect (JMP\*) instruction to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the physical unit table (\$PUT, section 2) is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

**WORD 10**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can write, word 10 contains the address used by I/O control when writing is requested of this I/O driver. If the I/O driver cannot write, word 10 is zero.

To write, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When writing is complete, the I/O driver returns control to I/O control with a JMP\* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

## **WORD 11**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can write an end of file, word 11 contains the address used by I/O control when a write-end-of-file is requested of this I/O driver. If the I/O driver cannot write an end of file, word 11 is zero.

To write an end of file, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the writing is complete, the I/O driver returns control to I/O control with a JMP\* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

## **WORD 12**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can rewind, word 12 contains the address used by I/O control when rewinding is requested of this I/O driver. If the I/O driver cannot rewind, word 12 is zero.

To rewind, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When rewinding is complete, the I/O driver returns control to I/O control with a JMP\* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

**WORD 13**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can skip files, word 13 contains the address used by I/O control when file-skipping is requested of this I/O driver. If the I/O driver cannot skip files, word 13 is zero.

To skip files, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the file-skipping is complete, the I/O driver returns control to I/O control with a JMP\* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

**WORD 14**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can skip records, word 14 contains the address used by the I/O control when record-skipping is requested of this I/O driver. If the I/O driver cannot perform a skip record function, word 14 is zero.

To skip records, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the record-skipping is complete, the I/O driver returns control to I/O control with a JMP\* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

**WORD 15**

|              |              |
|--------------|--------------|
| Modified by: | Not modified |
| Used by:     | I/O control  |

If the I/O driver can perform a function not otherwise covered, word 15 contains the address used by I/O control when a function is requested of this I/O driver. If the I/O driver cannot perform such a function, word 15 is zero.

## **user-coded I/O drivers**

To perform the function, I/O control passes to the I/O driver with the X register pointing to word 0 of the I/O driver DST. When the function is complete, the I/O driver returns control to I/O control with a JMP\* to the address in word 7.

The I/O driver sets a condition code in the A register before returning to I/O control. A negative A register indicates that more than 500 microseconds were spent in the I/O driver, and the PUT is scanned again because another I/O operation may have finished in the interim. A zero in the A register indicates that the I/O driver is busy. A positive A register (nonzero) indicates that the I/O driver is free.

| Peripheral Device                       | Device Name | Entry Name | Device Address |
|-----------------------------------------|-------------|------------|----------------|
| Magnetic tape<br>(controller 0, unit 0) | MT00        | \$00       | 010            |
| Magnetic tape<br>(controller 0, unit 1) | MT01        | \$01       | 010            |
| Magnetic tape<br>(controller 0, unit 2) | MT02        | \$02       | 010            |
| Magnetic tape<br>(controller 0, unit 3) | MT03        | \$03       | 010            |
| Magnetic tape<br>(controller 1, unit 0) | MT10        | \$04       | 011            |
| Magnetic tape<br>(controller 1, unit 1) | MT11        | \$05       | 011            |
| Magnetic tape<br>(controller 1, unit 2) | MT12        | \$06       | 011            |
| Magnetic tape<br>(controller 1, unit 3) | MT13        | \$07       | 011            |
| Magnetic tape<br>(controller 2, unit 0) | MT20        | \$08       | 012            |
| Magnetic tape<br>(controller 2, unit 1) | MT21        | \$09       | 012            |
| Magnetic tape<br>(controller 2, unit 2) | MT22        | \$0A       | 012            |
| Magnetic tape<br>(controller 2, unit 3) | MT23        | \$0B       | 012            |

Figure 12-1. I/O Drivers and Peripheral Devices (1 of 4)



| Peripheral Device                                         | Device Name | Entry Name | Device Address |
|-----------------------------------------------------------|-------------|------------|----------------|
| Magnetic tape<br>(controller 3, unit 0)                   | MT30        | \$0C       | 013            |
| Magnetic tape<br>(controller 3, unit 1)                   | MT31        | \$0D       | 013            |
| Magnetic tape<br>(controller 3, unit 2)                   | MT32        | \$0E       | 013            |
| Magnetic tape<br>(controller 3, unit 3)                   | MT33        | \$0F       | 013            |
| Teletype keyboard/<br>printer 1                           | TY00        | \$0G       | 001            |
| (controller 0, unit 0)<br>Teletype paper tape<br>punch 1  | TP00        | \$0H       | 001            |
| (controller 0, unit 0)<br>Teletype paper tape<br>reader 1 | TR00        | \$0I       | 001            |
| (controller 0, unit 0)<br>Teletype keyboard/<br>printer 2 | TY10        | \$0J       | ...            |
| (controller 1, unit 0)<br>Teletype paper tape<br>punch 2  | TP10        | \$0K       | ...            |
| (controller 1, unit 0)<br>Teletype paper tape<br>reader 2 | TP10        | \$0L       | ...            |

Figure 12-1. I/O Drivers and Peripheral Devices (2 of 4)

| Peripheral Device                                  | Device Name | Entry Name | Device Address |
|----------------------------------------------------|-------------|------------|----------------|
| Card reader 1                                      | CR00        | \$0M       | 030            |
| Card reader 2                                      | CR10        | \$0N       | ...            |
| High-speed paper tape reader/punch 1               | PT00        | \$0O       | 037            |
| High-speed paper tape reader/punch 2 (unformatted) | PT10        | \$0P       | 037            |
| Line printer 1                                     | LP00        | \$0Q       | 035            |
| Line printer 2                                     | LP10        | \$0R       | ...            |
| Card punch 1                                       | CP00        | \$0S       | 031            |
| Card punch 2                                       | CP10        | \$0T       | ...            |
| Drum (controller 0, unit 0)                        | DR00        | \$10       | 014            |
| .                                                  | .           | .          | .              |
| .                                                  | .           | .          | .              |
| Drum (controller 0, unit 0)                        | DR09        | \$19       | 014            |
| Disc (controller 0, unit 0)                        | DK00        | \$10       | 015            |
| (controller 0, unit 0)                             | .           | .          | (620-40/41)    |
| .                                                  | .           | .          | 016            |
| .                                                  | .           | .          | (620-39)       |
| Disc (controller 0, unit 0)                        | DK09        | \$19       |                |
| Disc (controller 0, unit 1)                        | DK10        | \$20       | 015            |
| (controller 0, unit 1)                             | .           | .          | (620-40/41)    |
| .                                                  | .           | .          | 016            |
| .                                                  | .           | .          | (620-39)       |
| Disc (controller 0, unit 1)                        | DK19        | \$29       |                |

Figure 12-1. I/O Drivers and Peripheral Devices (3 of 4)

|                                       |      |      |                    |
|---------------------------------------|------|------|--------------------|
| Disc<br>(controller 1, unit 0)        | DK40 | \$50 | 016<br>(620-40/41) |
| .                                     | .    | .    | 017                |
| .                                     | .    | .    | (620-39)           |
| Disc<br>(controller 1, unit 0)        | DK49 | \$59 |                    |
| First buffer interlace<br>controller  |      |      | 020-021            |
| Second buffer interlace<br>controller |      |      | 022-023            |
| Third buffer interlace<br>controller  |      |      | 024-025            |
| Fourth buffer interlace<br>controller |      |      | 026-027            |

Figure 12-1. I/O Drivers and Peripheral Devices (4 of 4)

# I/O DRIVER PROGRAMMING EXAMPLES

## Example 1

Output an alphanumeric character string to the Teletype:

|         | NAME | \$TY   |                        |
|---------|------|--------|------------------------|
| *       | DST  |        |                        |
| \$TY    | DATA | 0      | WORDS TRANSFERRED      |
|         | DATA | 0      | I/O STATUS             |
|         | DATA | 'TY99' | I/O DRIVER NAME        |
|         | DATA | -1     | I/O FLAG AND OP CODE   |
|         | DATA | 0      | COUNT PARAMETER        |
|         | DATA | 0      | LOCATION PARAMETER     |
|         | DATA | \$TCK  | CHECK I/O ADDRESS      |
|         | DATA | \$TCK  | CHECK I/O ADDRESS      |
|         | DATA | 0      | UNUSED                 |
|         | DATA | \$TWR  | ADDRESS OF WRITE I/O   |
|         | DATA | 0      | UNUSED                 |
|         | DATA | 0      | UNUSED                 |
|         | DATA | 0      | UNUSED                 |
|         | DATA | 0      | UNUSED                 |
|         | DATA | 0      | UNUSED                 |
| *       |      |        |                        |
| \$TCK   | DATA | 0      | TTY AVAILABILITY UNDE- |
|         |      |        | TERMINED, ASSUME READY |
|         | INCR | 1      |                        |
|         | JMP* | \$TCK  |                        |
| *       |      |        |                        |
| \$TWR   | LDA  | 5, 1   | GET COUNT PARAMETER    |
|         | STA  | 0, 1   | STORE IT IN WORD 0     |
|         | LDB  | 6, 1   | GET DATA LOCATION      |
|         | EXC  | 0401   | INITIALIZE TTY         |
|         | LDAI | 0201   |                        |
|         | JMPM | \$TOAR | OUTPUT PRINT ENABLE    |
| \$TWR 1 | LDA  | 5. 1   | GET COUNT              |
|         | DAR  |        |                        |
|         | JAN  | \$TWR2 | JMP IF END OF WORD     |

# user-coded I/O drivers

|        |      |          |                        |
|--------|------|----------|------------------------|
|        | STA  | 5,1      |                        |
|        | LDA  | 0,2      | GET DATA WORD          |
|        | LRLA | 8        |                        |
|        | JMPM | \$TOAR   | OUTPUT LEFT CHARACTER  |
|        | LRLA | 8        |                        |
|        | JMPM | \$TOAR   | OUTPUT RIGHT CHARACTER |
|        | IBR  |          |                        |
|        | JMP  | \$TWR1   |                        |
| \$TWR2 | LDAl | 0215     |                        |
|        | JMPM | \$TOAR   | OUTPUT CARRIAGE RETURN |
|        | LDAl | 0212     |                        |
|        | JMPM | \$TOAR   | OUTPUT LINE FEED       |
|        | LDAl | 0204     |                        |
|        | JMPM | \$TOAR   | OUTPUT PRINT OFF CHAR  |
|        | DECR | 1        | A REGISTER NEGATIVE    |
|        | STA  | 4,1      | TURN OFF REQUEST FLAG  |
| *      | JMP* | \$TCK    |                        |
| \$TOAR | DATA | 0        |                        |
|        | SEN  | 0101,++4 | WRITE REGISTER READY   |
|        | JMP  | *-2      | NOT READY, WAIT        |
|        | OAR  | 1        | OUTPUT A CHARACTER     |
|        | JMP* | \$TOAR   | RETURN                 |
|        | END  |          |                        |

## Example 2

Read binary records from magnetic tape unit 0 on controller 0, device address 010.

|       | NAME           | \$MT   |                     |
|-------|----------------|--------|---------------------|
| *     | DST            |        |                     |
| \$MT  | DATA           | 0      |                     |
|       | DATA           | 0      |                     |
|       | DATA           | 'MT99' |                     |
|       | DATA           | -1     |                     |
|       | DATA           | 0      |                     |
|       | DATA           | 0      |                     |
|       | DATA           | \$MTS  | CHECK I/O ADDRESS   |
|       | DATA           | \$MTS  | CHECK I/O ADDRESS   |
|       | DATA           | \$MRD  | ADDRESS OF READ I/O |
|       | DATA           | 0      |                     |
|       | DATA           | 0      |                     |
|       | DATA           | 0      |                     |
|       | DATA           | 0      |                     |
|       | DATA           | 0      |                     |
| *     | STATUS ROUTINE |        |                     |
| \$MTS | DATA           | 0      |                     |
|       | SEN            | 0210,A | IF MTU READY        |

|       |      |          |                          |
|-------|------|----------|--------------------------|
|       | TZA  |          | SET (AR) = 0             |
|       | JMP* | \$MTS    | RETURN (MTU BUSY)        |
| A     | LDA  | 1,1      | GET WORD 1 OF DST        |
|       | JAN  | B        | IF STATUS INVALID        |
|       | INCR | 01       | SET A = 1                |
|       | JMP* | \$MTS    | RETURN (MTU READY)       |
| B     | DECR | 01       | SET A = -1               |
|       | STA  | 4,1      | SET WORD 4 OF DST TO     |
|       |      |          | 'NOT BUSY'               |
|       | SEN  | 010,ERR  | IF ERROR OCCURRED        |
|       | SEN  | 0310,EOF | IF END OF FILE READ      |
|       | SEN  | 0510,EOT | IF END OF TAPE FOUND     |
|       | SEN  | 0610,BOT | IF BEGINNING OF TAPE     |
|       | JMP  | NORM     | INDICATE NORMAL STATUS   |
| BOT   | BSS  | 0        |                          |
| EOT   | IAR  |          | SET WORD 1 OF DST TO 3   |
| EOF   | IAR  |          | SET WORD 1 OF DST TO 2   |
| ERROR | IAR  |          | SET WORD 1 OF DST TO 1   |
| NORM  | IAR  |          | SET WORD 1 OF DST TO 0   |
|       | STA  | 1,1      |                          |
|       | JMP* | \$MTS    | RETURN, STATUS SET       |
| *     | READ | ROUTINE  |                          |
| \$MRD | LDB  | 6,1      | GET START MEMORY ADDR    |
|       | EXC  | 010      | START MTU                |
| C     | SEN  | 0110,D   | IF WORD READY            |
|       | SEN  | 0210,E   | IF MTU STOPPED           |
|       | JMP  | C        | WAIT                     |
| D     | CIA  | 010      | INPUT WORD               |
|       | STA  | 0,2      | STORE WORD               |
|       | IBR  |          | BUMP MEMORY POINTER      |
|       | INR  | 0,1      | INCREMENT NO OF XFERS    |
|       | LDA  | 5,1      |                          |
|       | DAR  |          |                          |
|       | STA  | 5,1      | DECREMENT COUNT          |
|       | JAZ  | E        | IF FINISHED              |
|       | JMP  | C        | CONTINUE                 |
| E     | LDAI | 0100004  |                          |
|       | STA  | 1,1      | STATUS INVALID, MTU BUSY |
|       | TZA  |          | (AR) = 0 FOR MTU BUSY    |
|       | JMP* | \$MTS    | RETURN                   |
|       | END  |          |                          |

## I/O SUPPORT SUBROUTINES

To prevent possible data loss when several I/O transfers are operating concurrently, MOS has a subroutine, IOOK, for determining if enough processor time is available before initiating an I/O operation. This subroutine is used by the I/O driver prior to data transfer. The subroutine calling sequence is:

JMPM

IOOK

Upon IOOK entry, the A register contains the I/O algorithm factor for that I/O device. The I/O algorithm factor depends on whether the device is operating in BIC or programmed data transfer mode. The equations for the two types of transfers are:

$$F_b = (T_b/T_t) + 0.1(T_b/T_t)$$

$$F_p = (N * C)/T_t + 0.1((N * C)/T_t)$$

where

F<sub>b</sub> = I/O algorithm factor for BIC transfers

F<sub>p</sub> = I/O algorithm factor for programmed transfers

C = CPU memory cycle time (microseconds)

T<sub>t</sub> = maximum transfer rate of I/O device (microseconds/word)

T<sub>b</sub> = maximum transfer rate of BIC (microseconds/word)

N = number of CPU memory cycles required by data transfer

On returning from IOOK, the I/O driver examines the A register. If the A register is positive, the I/O operation can be performed. If the A register is negative, more time is needed for the I/O operation than is available. In the latter case, the new I/O operation cannot start and the driver exits to I/O control indicating a busy device (A = 0).

Upon completion of a successful I/O operation, the I/O driver removes the algorithm factor from the system timing variable by again calling IOOK with the two's complement of the I/O algorithm factor in the A register.

### Example

If device A is operating with a BIC and has a data transfer rate of one word every 9.9 microseconds, and device B is operating with another BIC and has a data transfer rate of one word every 19.8 microseconds, can device C be operated over the I/O bus if it requires the following programmed transfers:

|     |     |           |                       |
|-----|-----|-----------|-----------------------|
| \$1 | SEN | DEVC, \$2 | DEVICE READY          |
|     | JMP | *-2       | NO, WAIT              |
| \$2 | CIA | DEVC      | YES, GET DATA         |
|     | STA | 0, 1      | STORE DATA IN ADDR(X) |

|      |      |                        |
|------|------|------------------------|
| INCR | 045  | INCREMENT DATA ADDRESS |
| SUB  | EADD |                        |
| JAN  | \$1  | JMP IF NOT END ADDRESS |

The I/O algorithm factor for device A is:  $(4.95/9.9) + 0.1(4.95/9.9) = 0.55$ . The I/O algorithm factor for device B is:  $(4.95/19.8) + 0.1(4.95/19.8) = 0.275$ . The I/O algorithm factor for device C is:  $(11.25 * 1.8)/81 + 0.1 ((11.25 * 1.8)/81) = 0.275$ . The sum of the I/O algorithm factors for devices A and B is 0.825, which is less than one. Thus, A and B use only about 82.5 percent of the available memory cycles. However, if C were added, the sum would be 1.1, requiring more memory cycles than available (110 percent). Therefore, C cannot be activated at this time and must wait for either A or B to complete its operation.

## I/O STATUS MESSAGES

The operator alert subroutine is called by the I/O driver to advise the operator that a peripheral device is not available. The calling sequence is:

|      |       |
|------|-------|
| JMPM | IOA\$ |
| DATA | DELY  |

The X register points to word 0 of the I/O driver. DELY is the address of a two-memory-word data block in the I/O driver.

IOA\$ examines and increments the first word of the DELY data block and returns to the calling routine if this word is negative. If the first word is positive, IOA\$ copies the contents of the second word into the first word and moves the name of the peripheral device from words 2 and 3 of the I/O driver DST into the message:

**xxxx - NOT READY**

This message is output at intervals of approximately 10 seconds to the Teletype printer by the resident message-printing subroutine. IOA\$ does not save any register contents when returning to the calling routine.



## BIC CONTROL

When an I/O driver is controlling a device that can operate on a BIC, subroutines BIR\$ and BIA\$ are used. In addition, a BIC control table is defined by the driver for each I/O device controller. A pointer to the table can be appended to the end of the DST for reference purposes. The BIC control subroutines used by the I/O driver have the same calling sequence:

|      |       |                                |
|------|-------|--------------------------------|
| CALL | BIR\$ | With the B register pointing   |
|      | BIA\$ | to word 0 of the control table |

## BIC CONTROL TABLE

The words in the BIC control table are defined as follows:

| Word | Definition                                                                                                              |
|------|-------------------------------------------------------------------------------------------------------------------------|
| 0    | Contains the BIC address associated with this device (figure 12-1). Set to -01 if none.                                 |
| 1    | Contains the BIC initial address set and used by BIR\$.                                                                 |
| 2    | Contains the abnormal BIC stop flag set by BIR\$. It is set after a BIC operation in which an abnormal BIC stop occurs. |
| 3    | Contains the word count (i.e., the number of words to be input or output through the BIC).                              |
| 4    | Contains the starting memory address for the BIC transfer.                                                              |
| 5-n  | Contains controller commands and flag information specific to each device.                                              |

The controller commands are built at assembly time with the appropriate device address (i.e., each controller has its own device address, hardwired).

The flag information is used by the driver to determine and monitor such things as the direction of a skip, the type of skip (records or files), etc.

#### NOTE

Both words 3 and 4 are used by the BIC routines to initiate an operation. These words should be initialized by the driver before BIA\$ is called.

### BIR\$

BIR\$ determines if there is a BIC associated with the control table by testing word 0 of the BIC control table. A negative value indicates that there is no BIC. BIR\$ returns and sets the A register negative. If word 0 is positive, BIR\$ uses the number to build the BIC instructions.

BIR\$ determines if the BIC is busy. If the BIC is busy, BIR\$ returns and sets the A register negative. If the BIC is not busy, BIR\$ continues.

BIR\$ determines if an abnormal BIC stop has occurred. If it has, BIR\$ returns and sets the A register negative. If not, BIR\$ returns and sets the A register positive.

#### NOTE

The contents of the X register are destroyed.

### BIA\$

BIA\$ determines if there is a BIC associated with the control table by testing word 0 of the BIC control table. A negative value indicates that there is no BIC. BIA\$ returns and sets the A register negative. If word 0 is positive, BIA\$ uses the number to build the BIC instructions.

BIA\$ initializes the BIC and outputs the initial address from word 4 of the BIC control table.

BIA\$ adds one less than the count from word 3 of the BIC control table to the initial address to obtain the final address, and then outputs it. BIA\$ activates the BIC and returns to the caller.

#### NOTE

The contents of the X register are destroyed.



## SECTION 13 – STATUS AND ERROR MESSAGES

### EXECUTIVE

During operation of the executive, the following status and error messages can be posted on the system output device:

| Message                                        | Cause                                                                | Action                                                                                                                                  |
|------------------------------------------------|----------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>ILL DIR</b>                                 | Invalid directive read by executive                                  | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>READ ERR ON SI INPUT</b><br>/ASSIGN SI = TY | I/O error while reading directive                                    | Using the TTY, manually input the directive that caused the error.                                                                      |
| /ASSIGN SI = TY                                | An end of file or end of device detected during reading on SI file   | System waits for operation action. The next directive should be entered through the TTY.                                                |
| <b>ASSIGN DIR PARAM ERR</b>                    | Either L or R parameter missing from an /ASSIGN                      | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>L = R INCOMPATIBLE</b>                      | R unit cannot perform the function required by the L unit on /ASSIGN | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |

| Message                                 | Cause                                                                                                         | Action                                                                                                                                  |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>CANNOT ASSIGN<br/>SI = DUM</b>       | Attempt to assign system input file to a dummy physical device                                                | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>L.U.xxxx MAY NOT<br/>BE REASSIGN</b> | Attempt to reassign SO or SF logical unit                                                                     | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>ILL LUN/NAME<br/>USED IN DIR</b>     | Invalid parameter on /IOLIST                                                                                  | List terminated. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB.   |
| <b>LUN.xx END/BEG<br/>OF DEV ERR</b>    | Physical end or beginning of device encountered on unit xx prior to completion of certain executive functions | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>CPY INPT UNIT<br/>UNABL TO READ</b>  | Invalid /COPYA or /COPYB (L unit cannot be read)                                                              | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>CPY OUTPT UNIT<br/>UNABL TO WRTE</b> | Invalid /COPYA or /COPYB (R unit cannot be written on)                                                        | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>CPY READ ERR</b>                     | I/O reading error during copying                                                                              | Copying terminated. Read next directive from SI.                                                                                        |

|                                         |                                                                                                                                                                                                        |                                                                                                                                         |
|-----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>CPY WRTE ERR</b>                     | I/O writing error during copying                                                                                                                                                                       | Copying terminated. Read next directive from SI.                                                                                        |
| <b>END OF CPY<br/>OUTPT DEV</b>         | Physical end of device encountered during copying                                                                                                                                                      | Copying terminated. Read next directive from SI.                                                                                        |
| <b>ILL LOAD/UPLOAD<br/>DIR PARAM xx</b> | Undefined parameters encountered during processing of /LOAD or /UPLOAD, where xx = first two characters of invalid parameter                                                                           | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>xx = N INVALID</b>                   | One or more of the following parameters out of range on /LOAD or /UPLOAD:<br>$0 \leq RP \leq \text{core size}$<br>$0 \leq RC \leq \text{core size}$<br>$0 \leq RI \leq 0777$<br>$0 \leq RL \leq 03777$ | No loading attempted.                                                                                                                   |
| <b>ILL F/A DIR<br/>PARAM</b>            | Invalid parameter on assembler or FORTRAN control directive                                                                                                                                            | Directive ignored. If SI = TY00, waits for input of next directive. If SI $\neq$ TY00, reads but ignores SI until next /JOB or /ENDJOB. |
| <b>NO/EOF, NO LOAD<br/>ATTMPTD</b>      | No /EOF after an assembly or compilation, therefore, no end of file on GO file.                                                                                                                        | No loading attempted                                                                                                                    |

## SYSTEM LOADER

During loading, the events listed abort the loading procedure. In this case, if SI = TY00, MOS waits for the next directive. If SI  $\neq$  TY00, SI is read but ignored until the next /JOB or /ENDJOB. When a map is requested, it is output prior to the termination message. For size and missing program errors, the names of all programs causing the error are also listed.

| Message             | Cause                                                                                                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| LDR READ ERR        | The loader encountered a reading error while attempting input of the object tape.                                                       |
| LDR RECORD ERR      | The loader input an invalid type of record.                                                                                             |
| LDR CKSM ERR        | The loader input an object text record with an invalid check-sum.                                                                       |
| LDR SEQ ERR         | The loader input an object text record with an invalid sequence number.                                                                 |
| LDR TEXT ERR        | The program object text contains an illegal or erroneous loader code.                                                                   |
| LDR DATA ERR        | The program attempted to overlay the loader, loader tables, or resident program; or the assembler attempted to overlay its I/O drivers. |
| LDR SIZE ERR        | Program memory requirements exceed available program/common storage.                                                                    |
| LIT POOL OVRFLW ERR | Program literal requirements exceed available literal storage.                                                                          |
| LDR COMMON ERR      | The programs contain conflicting size definitions for a common block.                                                                   |
| MISSG PGRMS ERR     | Loading requested named programs not found on either the binary or system file input devices.                                           |
| LDR INITZTN ERR     | Errors during loading of system loader.                                                                                                 |
| NO EXCTN ADDR       | Address to begin execution of the loaded program is missing from the object program.                                                    |

## I/O CONTROL

During program execution, calls to I/O control for input/output functions containing one of the following errors cause posting of a run-time error. In this case, if SI = TY00, MOS waits for the next directive. If SI  $\neq$  TY00, SI is read but ignored until the next /JOB or /ENDJOB.

| Message                        | Cause                                                                                              |
|--------------------------------|----------------------------------------------------------------------------------------------------|
| ILL LUN/NAME USED<br>IN DIR    | An undefined logical unit specified as a parameter in an I/O control directive.                    |
| IOCS CALL TO<br>UNASSGND LUNxx | An attempt to call an I/O driver not in memory (i.e., an unassigned logical unit specified by xx). |

## LANGUAGE PROCESSORS

During assembly and compilation, the following errors associated with language processors cause posting of an error message. In this case, if SI = TY00, MOS waits for the next directive. If SI  $\neq$  TY00, SI is read but ignored until the next /JOB or /ENDJOB.

| Message                        | Cause                                                                                            |
|--------------------------------|--------------------------------------------------------------------------------------------------|
| PROCSSR ERR JOB<br>ABORTD      | Language violation.                                                                              |
| PROCSSR RCRD COUNT<br>ERR      | The number of source statements read during pass 2 is not equal to that of pass 1.               |
| UN.xx SYS CNTRL DIR<br>NPT ERR | A control directive (i.e., a slash in column 1) was read from unit xx prior to an END statement. |
| UN.xx EOF ERR                  | An end of file was read from unit xx prior to an END statement.                                  |
| UN.xx END/BEG OF               | The physical end or beginning device was encountered on unit xx prior to an END statement.       |
| UN.xx I/O ERR                  | An unrecoverable I/O error occurred on unit xx during assembly or compilation.                   |



## DAS MR ASSEMBLER

During assembly, the source statements are checked for syntax errors and usage. In addition, errors can occur where the program cannot determine the correct meaning of the source statement.

When an error is detected, the assembler outputs an error code following the source statement containing the error, on the LO, and continues to the next statement.

The assembler error messages (codes) are:

| Code | Definition                                                                                                                                                                                                           |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| *AD  | An address expression is in error.                                                                                                                                                                                   |
| *DC  | A decimal character appears in an octal constant.                                                                                                                                                                    |
| *DD  | There is an invalid redefinition of a symbol or the location counter.                                                                                                                                                |
| *E   | The symbolic source statement is incorrectly formed.                                                                                                                                                                 |
| *EX  | An expression contains an illegal construction.                                                                                                                                                                      |
| *FA  | A floating point number contains a format error.                                                                                                                                                                     |
| *IL  | The first non-blank character of the source statement is invalid; the statement is not processed.                                                                                                                    |
| *NR  | No memory space available for the addition of an entry to the assembler's tables.                                                                                                                                    |
| *NS  | No symbol in the label field of a SET, EQU, MAC or FORM directive statement. No symbol in the label or variable field of an OPSY directive statement. No symbol in the variable field of a NAME directive statement. |

|     |                                                                                                                                                                                                                                          |
|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ⌘OP | The instruction field is undefined; two No Operation (NOP) instructions are generated in the object program. The remainder of the statement is not processed. Illegal nesting of DUP or MAC directive statements also causes this error. |
| ⌘QQ | Illegal use of prime (').                                                                                                                                                                                                                |
| ⌘R  | A relocatable item was encountered in the place where an absolute item was expected.                                                                                                                                                     |
| ⌘SE | The symbol in the location field has a value during pass 2 that is different than the value used in pass 1.                                                                                                                              |
| ⌘SY | An expression contains an undefined symbol.                                                                                                                                                                                              |
| ⌘SZ | The expression value is too large for the size of the subfield or a DUP statement specifies that more than three symbolic source statements are to be assembled.                                                                         |
| ⌘TF | Undefined or illegal index register specification.                                                                                                                                                                                       |
| ⌘UC | An undefined character in an arithmetic expression.                                                                                                                                                                                      |
| ⌘UD | Undefined symbol in variable field of USE directive.                                                                                                                                                                                     |
| ⌘XR | Address out of range for indexing specification.                                                                                                                                                                                         |
| ⌘ = | Illegal use of a literal.                                                                                                                                                                                                                |

## FORTRAN IV COMPILER AND RUNTIME

### COMPILER

During compilation, source statements are checked for such items as validity, syntax, and usage. When an error is detected, it is posted on the LO beneath the source statement. The errors marked T terminate binary output.

All error messages are of the form

**ERR xx c(1)-c(16)**

where xx is a number from 0 to 18 (notification error), or T followed by a number from 0 to 9 (terminating error); and c(1)-c(16) is the last character string (up to 16) encountered in the statement being processed. The right-most character indicates the point of error and the @ indicates the end of the statement. The possible error messages are:

| Notification Error | Definition                                |
|--------------------|-------------------------------------------|
| 0                  | Illegal character input                   |
| 1                  | Construction error                        |
| 2                  | Usage error                               |
| 3                  | Mode error                                |
| 4                  | Illegal DO termination                    |
| 5                  | Improper statement number                 |
| 6                  | Common base lowered                       |
| 7                  | Illegal equivalence group                 |
| 8                  | Reference to nonexecutable statement      |
| 9                  | No path to this statement                 |
| 10                 | Multiply defined statement number         |
| 11                 | Invalid format construction               |
| 12                 | Spelling error                            |
| 13                 | Format statement with no statement number |

| Terminating Error | Definition                        |
|-------------------|-----------------------------------|
| 14                | Function not used as variable     |
| 15                | Truncated value                   |
| 16                | Statement out of order            |
| 17                | More than 29 named common regions |
| 18                | Noncommon data                    |

| Terminating Error | Definition                |
|-------------------|---------------------------|
| T1                | Construction error        |
| T2                | Usage error               |
| T3                | Data pool overflow        |
| T4                | Illegal statement         |
| T5                | Improper use              |
| T6                | Improper statement number |
| T7                | Mode error                |
| T8                | Constant too large        |
| T9                | Improper DO nesting       |

## RUNTIME

When an error is detected during runtime, a message is posted on the SO device and the job is aborted. The messages and their definitions are:

| Message     | Cause                                                                                 |
|-------------|---------------------------------------------------------------------------------------|
| ARITH OVFL  | Arithmetic overflow                                                                   |
| GO TO RANGE | Computer GO TO out of range                                                           |
| FUNC ARG    | Invalid function argument (e.g., square root of negative number)                      |
| FORMAT      | Error in FORMAT statement                                                             |
| MODE        | Mode error (e.g., outputting real array with I format)                                |
| DATA        | Invalid input data (e.g., inputting a real number from external medium with I format) |
| I/O         | I/O error (e.g., parity, EOF)                                                         |

## FILE EDITING PROGRAM

During the file editing program, the following errors can occur:

| Message                                  | Cause                                                                                                                                                              | Action                                                                                                                           |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <b>INPUT FMT ERR n</b>                   | An unrecognizable record read from SI (n = 1) or from the source file, PI, (n = 2)                                                                                 | No action taken on PO.                                                                                                           |
| <b>CATL OFLO</b>                         | More than 20 entries in the catalog                                                                                                                                | The PO backspaces to the previous file, two EOFs are written and the job terminated.                                             |
| <b>OUTPUT EOM</b>                        | The end of media detected on PO                                                                                                                                    | Same as above.                                                                                                                   |
| <b>END INPUT-FILE filename NOT FOUND</b> | The second filename specified in a COPY control record not on the old source library (PI)                                                                          | Two EOFs are written on PO and the job terminated.                                                                               |
| <b>FILE filename NOT FOUND</b>           | The filename is an EDIT control record, or the first filename in a COPY control record is not on PI.                                                               | The control directive is ignored and the job continues.                                                                          |
| <b>CTRL REC ERROR n</b>                  | A control record is misplaced (n = 1), a required parameter is missing (n = 2), contains too many parameters (n = 3), or contains a parameter format error (n = 4) | If n = 1, action is described under INPUT FMT ERR n. If n = 2, 3, or 4, the control record is ignored and the program continues. |
| <b>END INPUT</b>                         | An EOF or MOS control                                                                                                                                              | Same as for INPUT FMT                                                                                                            |

| Message                           | Cause                                                                                                                        | Action                                                                                                                                                                                                                                                                                                                                                                      |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                   | directive (/card) is read when a file editing control directive was expected.                                                | ERR n.                                                                                                                                                                                                                                                                                                                                                                      |
| SEQ. OFLO                         | While resequencing a file, more than 9,999 data records are read.                                                            | Sequencing restarts at zero and the program continues.<br><br>NOTE: The records in this file can no longer be accessed by internal sequence numbers.                                                                                                                                                                                                                        |
| aaannnn NOT FOUND                 | While searching a file by internal sequence number, a sequence number higher than the one being searched for is encountered. | If the desired sequence number is the first number of a DEL or RPL group, or is the sequence number specified in an ADD control record, the higher number is used in its place.<br><br>If the desired sequence number is the second number of a DEL or RPL group, the record preceding the record containing the higher number is taken to be the last record of the group. |
| SEQ. ERROR<br>aaannnn-<br>aaannnn | While searching a file by internal sequence number, an out-of-sequence condition is found.                                   | Same as for INPUT FMT ERR n.                                                                                                                                                                                                                                                                                                                                                |
| CTRL. SEQ.<br>ERROR               | An internal sequence number or external line number lower than the previous one is in an ADD, DEL, or CHG control record.    | Same as for INPUT FMT ERR n.                                                                                                                                                                                                                                                                                                                                                |

## SYSTEM MAINTENANCE PROGRAM

During the system maintenance operation, the following errors can occur:

| Message                          | Cause                                                                                                            |
|----------------------------------|------------------------------------------------------------------------------------------------------------------|
| <b>CHECKSUM ERROR</b> name       | There is a checksum error in a record of the named program.                                                      |
| <b>READ ERROR</b> name           | An error indication is received from I/O control after reading a record of the named program.                    |
| <b>WRITE ERROR</b> name          | An error indication is received from I/O control after writing a record of the named program.                    |
| <b>NO NAME</b> name              | The named program contains no entry name in the binary object module.                                            |
| <b>RECORD SIZE</b> name          | The size of a binary record as determined by I/O control for the named program is not 60 words (53 for the 622). |
| <b>LOADER CODE ERROR</b><br>name | The binary loader text for the named program contains a code or subcode not recognized by the loader.            |
| <b>SEQUENCE ERROR</b> name       | A sequence number in the binary object module for the named program is incorrect.                                |
| <b>STRUCTURE ERROR</b> name      | There is a nonbinary record in the named object module.                                                          |

After an error message is logged, enter one of the following statements:

| Statement    | Definition                                                                                                                                                                                               |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>RCRD</b>  | Reread the last record. If the error occurred on a magnetic tape, drum, or disc unit, the system maintenance program backspaces the record. Otherwise, manually position the record so it can be reread. |
| <b>PGRM</b>  | Restart the program. If the error occurred on a magnetic tape, drum, or disc unit, the program backspaces to the beginning of the program. Otherwise, manually position the program so it can be reread. |
| <b>SMAIN</b> | Restart the system maintenance program.                                                                                                                                                                  |

Key-in errors result in a repeat of the error message:

The system then waits for a correct entry.



## SYSTEM PREPARATION

During system preparation, the following errors can occur. The first two can be corrected by reentering the information noted.

| Message                                                         | Cause                                                       | Action                                 |
|-----------------------------------------------------------------|-------------------------------------------------------------|----------------------------------------|
| <b>ILLEGAL PREP<br/>DIRECTIVE</b>                               | There is a syntax error in the control directive just read. | Reenter the directive through the TTY. |
| <b>DEVICE NAME NOT<br/>VALID ENTER DE-<br/>VICE NAME FOR xx</b> | There is a syntax error in a peripheral device name.        | Reenter the name through the TTY.      |

The errors listed below are corrected by the procedure given at the end of the list.

| Message                                | Cause                                                                                                                                    |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <b>SEQUENCE ERROR name</b>             | A sequence number in the object module for the named program is incorrect.                                                               |
| <b>STRUCTURE ERROR name</b>            | There is a nonbinary record in the named object module.                                                                                  |
| <b>LITERAL POOL OVERFLOW<br/>name</b>  | The size allocated to the literal pools is exceeded during the generation of the named absolute module.                                  |
| <b>COMMON ERROR name</b>               | The size of a common block is illegally redefined during the generation of the named absolute program.                                   |
| <b>PROGRAM SIZE ERROR<br/>name</b>     | The size of the named absolute program exceeds the available memory.                                                                     |
| <b>MEMORY SIZE ERROR<br/>name</b>      | The tables internal to the system preparation program during the generation of the absolute program named exceed the available memory.   |
| <b>MISSING PROGRAMS<br/>ERROR name</b> | During the generation of the absolute program named, there is an external reference that cannot be satisfied before encountering ENDABS. |

|                                    |                                                                                                                                    |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <b>NO EXECUTE ADDRESS<br/>name</b> | No program execution address is found for any program during the generation of the absolute program named.                         |
| <b>CHECKSUM ERROR name</b>         | There is a check-sum error in a record of the named program.                                                                       |
| <b>READ ERROR name</b>             | An error, end, or beginning of device is received from I/O control after a reading operation for the named program.                |
| <b>WRITE ERROR name</b>            | An error, end or beginning of device, or end of file is received from I/O control after a writing operation for the named program. |
| <b>NO NAME name</b>                | The named program contains no name in the object module.                                                                           |
| <b>LOADER CODE ERROR<br/>name</b>  | The object module for the named program contains a code or subcode not recognized by the loader.                                   |

After an error message is logged, enter one of the following statements:

| <b>Statement</b> | <b>Definition</b>                                                                                                                                                                                                                                                         |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>RCRD</b>      | Reread the last record. If the error occurred on a magnetic tape, drum, or disc unit, the system preparation program backspaces the record. Otherwise, manually position the record so it can be reread. This procedure does not apply to errors during ABS processing.   |
| <b>PGRM</b>      | Restart at the beginning of the current binary loader text program. If the error occurred on a magnetic tape, drum or disc unit, the system preparation program backspaces to the beginning of the program. Otherwise, manually position the program so it can be reread. |

Key-in errors result in a repeat of the error message:

The system then waits for a correct entry.



## APPENDIX A — ABSOLUTE MODULE FORMAT

Absolute modules are created by the system preparation program to construct the system file. The modules are made up of text and string records.

**Word 1** of each record is a control word, as follows:

| Bit   | Binary Value | Definition                                                                               |
|-------|--------------|------------------------------------------------------------------------------------------|
| 16-17 | --           | 622-series computers only: unused                                                        |
| 15    | 0            | Verify the record check-sum                                                              |
|       | 1            | Suppress check-sum verification                                                          |
| 13-14 | 11           | Binary record                                                                            |
|       | 00-10        | Nonbinary record                                                                         |
| 12    | 0            | First record of a module                                                                 |
|       | 1            | Not the first record of a module                                                         |
| 11    | 0            | Last record of a module -- execution address in the word following the last word of text |
|       | 1            | Not the last record of a module                                                          |
| 10    | 0            | Text record                                                                              |
|       | 1            | String record                                                                            |
| 9     | --           | Unused                                                                                   |
| 8     | 0            | Absolute module                                                                          |
|       | 1            | Not an absolute module (e.g., relocatable object module)                                 |
| 0-7   | Any          | Sequence number                                                                          |

**Word 2** contains the exclusive-OR check-sum of words 1 and 3 through 60.

**Word 3** contains zeros when the first record of a module. When not the first record, it contains text.

**Words 4 through 7** in the first record of a module contain the module (program) name as left-justified, blank-filled ASCII characters. When not in the first record, these words contain text.

**Words 8 through 11** in the first record contain the creation date of the module (program) in ASCII characters. When not in the first record, these words contain text.

**Words 12 through 60** contain text.

Text data on text records (word 1, bit 10 = 0) is organized into a variable number of variable length blocks as follows:

| Word       | Contents                                                    |
|------------|-------------------------------------------------------------|
| 3          | Number of words of text that follow (n-1) in block 1        |
| 4          | Starting load location for the text that follows in block 1 |
| 5          | Text word 1 in block 1                                      |
| .          | .                                                           |
| .          | .                                                           |
| .          | .                                                           |
| 4 + n      | Text word n in block 1                                      |
| .          | .                                                           |
| 5 + n      | Number of words of text that follow (n-1) in block 2        |
| 6-n        | Starting load location for text that follows in block 2     |
| 7-n        | Text word 1 in block 2                                      |
| .          | .                                                           |
| .          | .                                                           |
| .          | .                                                           |
| 6 + n + n' | Text word n in block 2                                      |
| .          | .                                                           |
| .          | .                                                           |
| .          | .                                                           |

A text record can contain a number of text words equal to or less than the capacity of one card, i.e., up to 57 words. However, it will never contain a partial block. When a record is not full, the last *number of words of the text minus one* value is negative to indicate the logical end of the record.

String data on string records (word 1, bit 10 = 1) is organized into a variable number of variable length blocks of information, as follows:

| Word   | Contents                                                               |
|--------|------------------------------------------------------------------------|
| 3      | Number of words of string entry points minus one (n-1) on this record. |
| 4      | String entry point 1                                                   |
| 5      | String reference 1, 1                                                  |
| .      | .                                                                      |
| .      | .                                                                      |
| .      | .                                                                      |
| 4-m    | One's complement of string reference 1,m                               |
| .      | .                                                                      |
| .      | .                                                                      |
| .      | .                                                                      |
| 5-m    | String entry point 2                                                   |
| 6-m    | String reference 2,1                                                   |
| .      | .                                                                      |
| .      | .                                                                      |
| .      | .                                                                      |
| 6-m-m' | One's complement of string reference 2,m                               |

## **APPENDIX B — OBJECT MODULE FORMAT**

Object modules generated by the MOS language processors result from assembly or compilation. The modules are input by the MOS loader and are bound together into an executable program.

The first record of the module contains the size of the program, an eight-character identification, and an eight-character date. Entry name addresses, if any, appear as the first data field items of the object module.

### **RECORD STRUCTURE**

#### **Sixteen-Bit Computers**

Object module records have a fixed length of sixty 16-bit words. Word 1 is the record control word. Word 2 contains the exclusive-OR check-sum of word 1 and words 3 to 60. Words 3 to 11 can contain a program identification block (optional). Words 12 to 60 (or 3 to 60 if there is no program identification block) contain data fields.

#### **Eighteen-Bit Computers**

Object module records have a fixed length of fifty-three 18-bit words. Word 1 is the record control word. Word 2 contains the exclusive-OR check-sum of word 1 and words 3 through 53. Words 3 through 11 can contain a program identification block (optional). Words 12 through 53 (or 3 through 53 if there is no program identification block) contain data fields.

### **PROGRAM IDENTIFICATION BLOCK**

The program identification (ID) block appears in words 3 to 11 of the starting record of each module. Word 3 contains the program size, words 4 to 7 contain an ASCII eight-character program identification, and words 8 to 11 contain an ASCII eight-character date.

### **DATA FIELD FORMATS**

Data fields contain one-, two-, three-, or four-word entries. One-word entries consist of a control word; two-word entries consist of a control word and a data word; three-word entries consist of a control word and two data words; and four-word entries consist of a control word, two name words, and a data word. Data words can contain instructions, constants, chain addresses, entry addresses, and address offset values.

| Bit   | Binary Value | Meaning                                   |
|-------|--------------|-------------------------------------------|
| 16-17 | 00           | Undefined (18-bit computers only)         |
| 15    | 0            | Verify check-sum                          |
|       | 1            | Suppress check-sum                        |
| 13-14 | 11           | Binary record                             |
|       | 00-10        | Nonbinary record                          |
| 12    | 0            | First record of module                    |
|       | 1            | Not the first record                      |
| 11    | 0            | Last record of module                     |
|       | 1            | Not the last record                       |
| 10    | 0            |                                           |
| 9     | 0            |                                           |
| 8     | 0            | Not a relocatable module (i.e., absolute) |
|       | 1            | Relocatable module                        |
| 0-7   |              | Sequence number (modulo 255)              |

Figure A-1. Record Control Word Format

## LOADER CODES

Loader codes, which have the following format, are among the data in an object module.

|       |    |    |       |    |    |         |    |   |         |   |   |       |   |   |       |   |   |
|-------|----|----|-------|----|----|---------|----|---|---------|---|---|-------|---|---|-------|---|---|
| 17    | 16 | 15 | 14    | 13 | 12 | 11      | 10 | 9 | 8       | 7 | 6 | 5     | 4 | 3 | 2     | 1 | 0 |
| ----- |    |    | ----- |    |    | -----   |    |   | -----   |   |   | ----- |   |   | ----- |   |   |
| 622/i |    |    | Code  |    |    | Subcode |    |   | Pointer |   |   | Name  |   |   |       |   |   |

| Code  | Values | Meaning                                                                                                                                                                                                                                                                                       |
|-------|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 00    |        | Refer to subcode for specific action.                                                                                                                                                                                                                                                         |
| 01    |        | Undefined.                                                                                                                                                                                                                                                                                    |
| 02    |        | Add the value of the selected pointer to the data word before loading.                                                                                                                                                                                                                        |
| 03    |        | Add the value of the selected pointer to the first data word (literal value) and enter the sum in the direct literal pool if bit 11 of the second data word is zero. Otherwise, enter it in the indirect literal pool. Add the address of the literal to the second data word before loading. |
| 04    |        | Load the data word(s) absolute. Bits 12 through 0 indicate the number of words minus one (n-1) to load.                                                                                                                                                                                       |
| 05-07 |        | Undefined.                                                                                                                                                                                                                                                                                    |

| Subcode | Values | Meaning                                                                                                                                                                                                     |
|---------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 00      |        | Ignore this entry (one word only).                                                                                                                                                                          |
| 01      |        | Set the loading address counter to the sum of the specified pointer plus the data word.                                                                                                                     |
| 02      |        | Chain the current loading address counter value to the chain whose last address is given by the sum of the selected pointer plus the data word. Stop chaining when an absolute zero address is encountered. |
| 03      |        | Complete the postprogram references by adding to each address the sum of the selected pointer plus the data word.                                                                                           |
| 04      |        | Undefined.                                                                                                                                                                                                  |



## object module format

|         |                                                                                                                                                                                               |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 05      | Load the I/O driver currently associated with the logical device number specified by the sum of the selected pointer plus the data word.                                                      |
| 06      | Undefined.                                                                                                                                                                                    |
| 07      | Set the program execution address to the sum of the values of the selected pointer plus the data word.                                                                                        |
| 010     | Define the entry name with entry location as equal to the value of the selected pointer plus the data word.                                                                                   |
| 011     | Define a region for the pointer whose size is given in the data word. If the entry name is not blank, define the entry point as the base of the region.                                       |
| 012     | Enter a load request for the external name. The chain address is given by the sum of the selected pointer plus the data word.                                                                 |
| 013     | Enter the loading address of the external name in the indirect literal pool. Add the address of the literal plus the value of the selected pointer to the data word (command) before loading. |
| 014-017 | Undefined.                                                                                                                                                                                    |

| Pointer Values | Meaning |
|----------------|---------|
|----------------|---------|

|        |                           |
|--------|---------------------------|
| 00     | Program region.           |
| 01     | Postprogram region.       |
| 02     | Blank common region.      |
| 03-036 | Labelled COMMON regions.  |
| 037    | Absolute (no relocation). |

### Name Format

Names are one to six (six-bit) characters, starting in bit 3 of the control word and ending with bit 0 of the second name word. Only the right 16 bits of the two name words are used.

## EXAMPLE

The following is a sample program with description of the object module format after assembly and the core image after loading.

**Source Module**

|             |             |             |
|-------------|-------------|-------------|
|             | <b>NAME</b> | <b>SUBR</b> |
|             | <b>EXT</b>  | <b>BBEN</b> |
| <b>SUBR</b> | <b>ENTR</b> |             |
|             | <b>LDA*</b> | <b>SUBR</b> |
|             | <b>CALL</b> | <b>BBEN</b> |
|             | <b>STA</b>  | <b>TIME</b> |
|             | <b>JAN</b>  | <b>DONG</b> |
|             | <b>LDA</b>  | <b>=2</b>   |
|             | <b>CALL</b> | <b>BBEN</b> |
| <b>DONG</b> | <b>INR</b>  | <b>SUBR</b> |
|             | <b>JMP*</b> | <b>SUBR</b> |
| <b>TIME</b> | <b>BSS</b>  | <b>I</b>    |
|             | <b>END</b>  |             |

**Object Module**

060400      Record control word (first and last record, verify check-sum sequence number 0)

157631      Check-sum word.

             (Begin program ID block)

000016      Program size (exclusive of FORTRAN COMMON, literals, and indirect address pointers).

142730      Identification in ASCII (assume this program was labeled  
140715      EXAMPLE).

150314

142640

131263      Date of creation in ASCII (assume assembled 03-10-69)

126661

130255

133271

             (End program ID block)

010000      Define entry name SUBR at relative 0 (code 0, subcode 010,  
000647      pointer 0, name SUBR, and data word 0).

054262

000000

100000      Enter absolute data word 0 in memory at relative 0.

000000

## object module format

|        |                                                               |
|--------|---------------------------------------------------------------|
| 060000 | Enter literal (indirectly addressed relative 0) in indirect   |
| 100000 | pointer pool, add address of pointer to load 017000 and enter |
| 017000 | memory at relative 1.                                         |
|        |                                                               |
| 100000 | Enter absolute data word 02000 in memory at relative 2.       |
| 002000 |                                                               |
|        |                                                               |
| 100000 | Enter absolute data word 000000 in memory at relative 3.      |
| 000000 |                                                               |
|        |                                                               |
| 100000 | Enter absolute data word 054010 in memory at relative 4.      |
| 054010 |                                                               |
|        |                                                               |
| 100000 | Enter absolute data word 01004 in memory at relative 5.       |
| 001004 |                                                               |
|        |                                                               |
| 040000 | Enter relative data word 012 in memory at relative 6.         |
| 000012 |                                                               |
|        |                                                               |
| 060760 | Enter literal (absolute 2) into literal pool, add address     |
| 000002 | of literal to load command 010000, and enter in memory at     |
| 010000 | relative 7.                                                   |
|        |                                                               |
| 100000 | Enter absolute data word 02000 in memory at relative 010.     |
| 002000 |                                                               |
|        |                                                               |
| 040000 | Enter relative data word 03 in memory at relative 011.        |
| 000003 |                                                               |
|        |                                                               |
| 060000 | Enter literal (relative 0) into indirect pointer pool, add    |
| 000000 | address of literal to increment command 047000, and enter     |
| 047000 | in memory at relative 012.                                    |
|        |                                                               |
| 100000 | Enter absolute data word 01000 in memory at relative 013.     |
| 001000 |                                                               |
|        |                                                               |
| 040000 | Enter relative data word 0100000 in memory at relative 014.   |
| 100000 |                                                               |
|        |                                                               |
| 001000 | Set loading location for next command, if any, to relative    |
|        | 016.                                                          |
|        |                                                               |
| 012003 | Enter load request for external name BBEN and chain entry     |
| 000212 | address to relative 011.                                      |
| 024556 | 000011                                                        |

(The remaining words of this record contain zero.)

## Core Image

Assume the program originates at 0500, the literal pool limits are 0200-0400, and BBEN is loaded at 0516.

|      |        |      |      |
|------|--------|------|------|
| 0200 | 100500 | DATA | 0500 |
| 0201 | 000500 | DATA | 0500 |
| .    |        |      |      |
| .    |        |      |      |
| .    |        |      |      |
| 0377 | 000002 | DATA | 2    |
| .    |        |      |      |
| .    |        |      |      |
| .    |        |      |      |
| 0500 | 000000 | ENTR | 0    |
| 0501 | 017200 | LDA* | 0200 |
| 0502 | 002000 | JMPM |      |
| 0503 | 000516 |      | 516  |
| 0504 | 054010 | STA  | 0515 |
| 0505 | 001004 | JAN  |      |
| 0506 | 000512 |      | 0512 |
| 0507 | 010377 | LDA  | 0377 |
| 0510 | 002000 | JMPM |      |
| 0511 | 000516 |      | 0516 |
| 0512 | 047201 | INR* | 0201 |
| 0513 | 001000 | JMP  |      |
| 0514 | 100500 | *    | 0500 |
| 0515 |        | BSS  | 1    |
| 0516 |        | BSS  | 1    |

## object module format

The following six-bit codes are used by the loader in building object modules. The codes define names created by NAME, EXT, and /JOB directives.

| Character | Octal | Character | Octal | Character | Octal |
|-----------|-------|-----------|-------|-----------|-------|
| @         | 40    | V         | 66    | +         | 13    |
| A         | 41    | W         | 67    | ,         | 14    |
| B         | 42    | X         | 70    | -         | 15    |
| C         | 43    | Y         | 71    | .         | 16    |
| D         | 44    | Z         | 72    | /         | 17    |
| E         | 45    | [         | 73    | 0         | 20    |
| F         | 46    | \         | 74    | 1         | 21    |
| G         | 47    | ]         | 75    | 2         | 22    |
| H         | 50    | †         | 76    | 3         | 23    |
| I         | 51    | —         | 77    | 4         | 24    |
| J         | 52    | (blank)   | 00    | 5         | 25    |
| K         | 53    |           | 01    | 6         | 26    |
| L         | 54    | -         | 02    | 7         | 27    |
| M         | 55    |           | 03    | 8         | 30    |
| N         | 56    | \$        | 04    | 9         | 31    |
| O         | 57    |           | 05    | :         | 32    |
| P         | 60    | &         | 06    | ;         | 33    |
| Q         | 61    | '         | 07    | <         | 34    |
| R         | 62    | (         | 10    | =         | 35    |
| S         | 63    | )         | 11    | >         | 36    |
| T         | 64    | *         | 12    |           | 37    |
| U         | 65    |           |       |           |       |

## APPENDIX C – DATA FORMAT

This appendix explains the formats and symbols used by MOS for storing information on cards and paper tape.

### PAPER TAPE

Information stored on paper tape is either binary or alphanumeric. It is separated into records (blocks of words) by three blank frames. The last frame of each record contains an end-of-record mark (1-3-4-8 punch).

#### Binary Mode

Binary information is stored with three frames per computer word (Figure A-2). Note that channels 6 and 7 are always punched.

#### Alphanumeric Mode

Alphanumeric and unformatted binary information is stored with one frame per character or one frame per 8 bits of unformatted binary data (Figure A-3). Standard ASCII-8 punch levels are used.

#### Special Characters

An end of file is represented by the ASCII-8 BELL character (1-2-3-8 punch).

When paper tape is punched on a teletypewriter, the ASCII-8 ERROR character flags erroneous frames punched by the teletypewriter when it is turned on or off. This notifies the TTY and paper tape reader drivers to ignore the next frame.

When alphanumeric input tapes are punched off-line on a teletypewriter, there is no means of spacing the three blank frames after every record. The following procedure gives a tape that can be read by the TTY reader and paper tape reader drivers:

- a. Punch the alphanumeric statement.
- b. Punch an end of record (RETURN on the TTY keyboard).
- c. Punch three or more frames containing any of the following characters:

#### Press CONTROL and:

@  
LINE FEED  
WRU  
EOT  
RU  
VT

#### ASCII-8 Equivalent

DCo  
LINE FEED  
WRU  
EOT  
RU  
V. TAB

TAB  
HERE IS (33 ASR only)

H. TAB  
NULL

### **NOTE**

Any of these characters can also be used for leader and trailer.

- d. Punch the next alphanumeric statement. Return to step b.

## **CARDS**

Information stored on cards is either binary or alphanumeric. Each card holds one record of information. Hence, there is no end-of-record character for cards.

### **Binary Mode**

Binary information is stored with sixty 16-bit words or fifty-three 18-bit words per card. The information is serial with bit 15 of the first word in row 12 of column 1, bit 14 in row 11, etc. (figure A-4). Note that 18-bit records occupy only the first 954 bits on the card (i.e., the last six bits in column 80 are not used).

### **Alphanumeric Mode**

Alphanumeric information is stored one character per card column (figure A-5) using the standard punch patterns.

### **Special Character**

An end of file is represented on cards by a 2-7-8-9 punch in column 1 of an otherwise blank card.

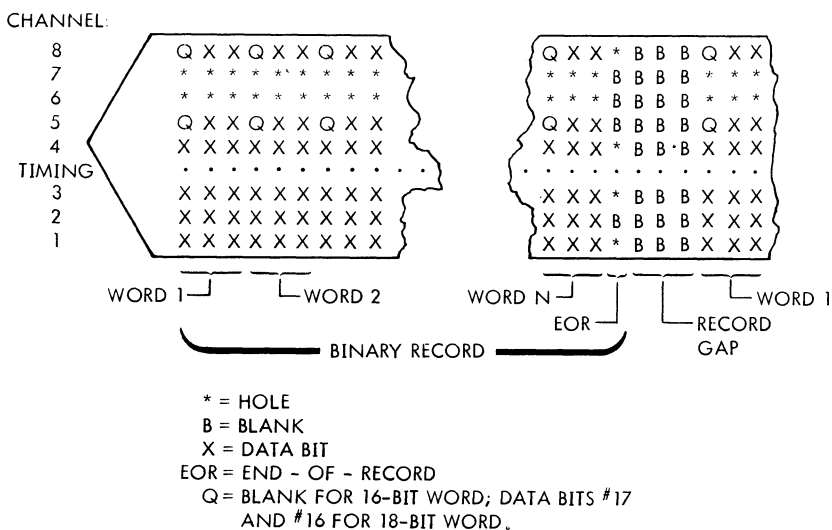


Figure A-2. Paper Tape Binary Record Format



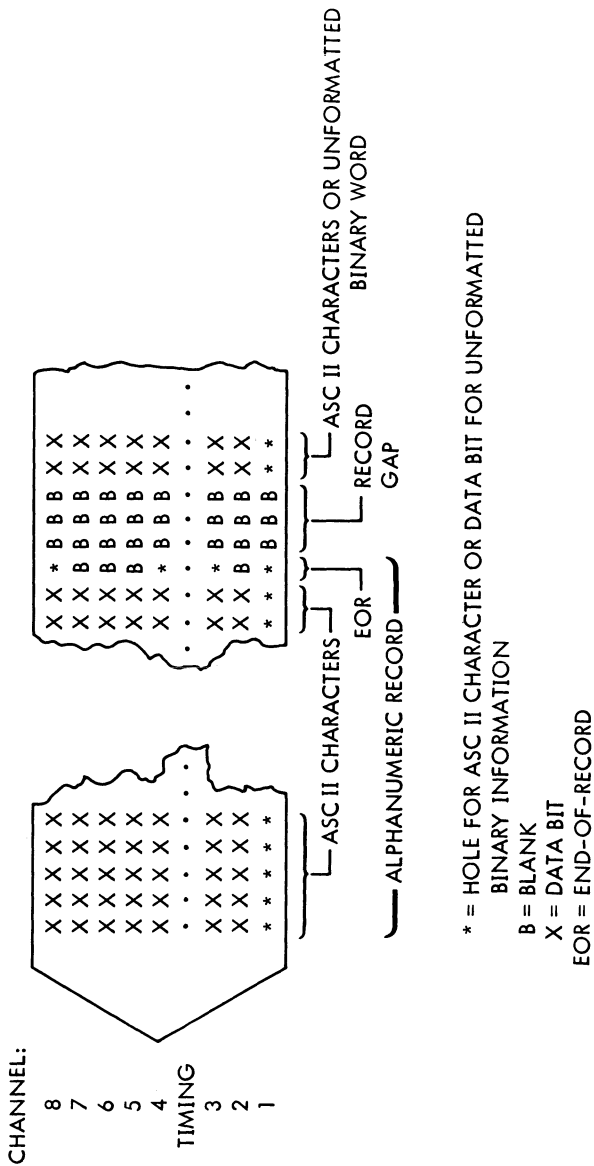
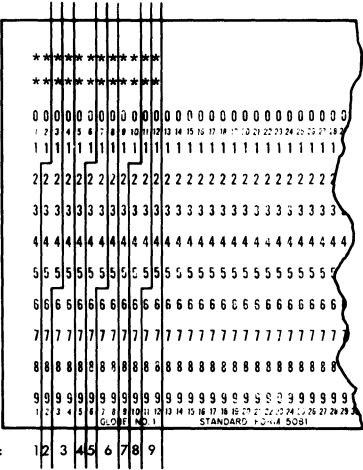
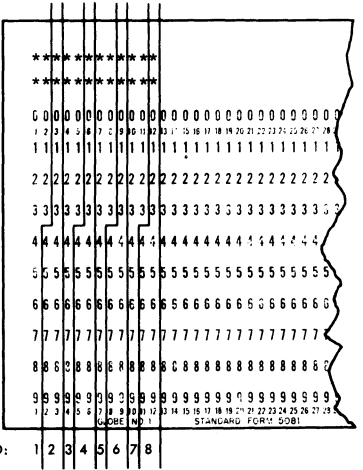


Figure A-3. Paper Tape Alphabetic Record Format



A. 16-BIT WORD FORMAT



B. 18-BIT WORD FORMAT

Figure A-4. Card Binary Record Format



**Figure A-5. Card Alphanumeric Record Format (IBM 026)**

## APPENDIX D – STANDARD CHARACTER CODES

| Print<br>Character | Printer | Mag Tape<br>(7 trk. BCD) | ASCII | FORTTRAN<br>(internal) | Hollerith<br>026 | 029      |
|--------------------|---------|--------------------------|-------|------------------------|------------------|----------|
| (blank)            | 040     | 20                       | 240   | 00                     | no punch         | no punch |
| !                  | 041     | 52                       | 241   | 51                     | 11-2-8           | 11-2-8   |
| "                  | 042     | 35                       | 242   | 62                     | 0-5-8            | 7-8      |
| #                  | 043     | 37                       | 243   | 63                     | 0-7-8            | 3-8      |
| \$                 | 044     | 53                       | 244   | 60                     | 11-3-8           | 11-3-8   |
| %                  | 045     | 57                       | 245   | 64                     | 11-7-8           | 0-4-8    |
| &                  | 046     | 77                       | 246   | 65                     | 12-7-8           | 12       |
| '                  | 047     | 14                       | 247   | 66                     | 4-8              | 5-8      |
| (                  | 050     | 34                       | 250   | 52                     | 0-4-8            | 12-5-8   |
| )                  | 051     | 74                       | 251   | 53                     | 12-4-8           | 11-5-8   |
| *                  | 052     | 54                       | 252   | 47                     | 11-4-8           | 11-4-8   |
| +                  | 053     | 60                       | 253   | 45                     | 12               | 12-6-8   |
| ,                  | 054     | 33                       | 254   | 54                     | 0-3-8            | 0-3-8    |
| -                  | 055     | 40                       | 255   | 46                     | 11               | 11       |
| .                  | 056     | 73                       | 256   | 51                     | 12-3-8           | 12-3-8   |
| /                  | 057     | 21                       | 257   | 50                     | 0-1              | 0-1      |
| 0                  | 060     | 12                       | 260   | 01                     | 0                | 0        |
| 1                  | 061     | 01                       | 261   | 02                     | 1                | 1        |
| 2                  | 062     | 02                       | 262   | 03                     | 2                | 2        |
| 3                  | 063     | 03                       | 263   | 04                     | 3                | 3        |
| 4                  | 064     | 04                       | 264   | 05                     | 4                | 4        |
| 5                  | 065     | 05                       | 265   | 06                     | 5                | 5        |
| 6                  | 066     | 06                       | 266   | 07                     | 6                | 6        |
| 7                  | 067     | 07                       | 267   | 10                     | 7                | 7        |
| 8                  | 070     | 10                       | 270   | 11                     | 8                | 8        |
| 9                  | 071     | 11                       | 271   | 12                     | 9                | 9        |

## APPENDIX D – STANDARD CHARACTER CODES

| Print<br>Character | Printer | Mag Tape<br>(7 trk. BCD) | ASCII | FORTTRAN<br>(internal) | Hollerith<br>026 | 029    |
|--------------------|---------|--------------------------|-------|------------------------|------------------|--------|
| :                  | 072     | 15                       | 272   | 67                     | 5-8              | 2-8    |
| ;                  | 073     | 56                       | 273   | 70                     | 11-6-8           | 11-6-8 |
| <                  | 074     | 76                       | 274   | 76 <sup>2</sup>        | 12-6-8           | 12-4-8 |
| =                  | 075     | 13                       | 275   | 55                     | 3-8              | 6-8    |
| >                  | 076     | 16                       | 276   | 76 <sup>4</sup>        | 6-8              | 0-6-8  |
| ?                  | 077     | 72                       | 277   | 76 <sup>2</sup>        | 12-2-8           | 0-7-8  |
| @                  | 100     | 32                       | 300   | 77                     | 0-2-8            | 4-8    |
| A                  | 101     | 61                       | 301   | 13                     | 12-1             | 12-1   |
| B                  | 102     | 62                       | 302   | 14                     | 12-2             | 12-2   |
| C                  | 103     | 63                       | 303   | 15                     | 12-3             | 12-3   |
| D                  | 104     | 64                       | 304   | 16                     | 12-4             | 12-4   |
| E                  | 105     | 65                       | 305   | 17                     | 12-5             | 12-5   |
| F                  | 106     | 66                       | 306   | 20                     | 12-6             | 12-6   |
| G                  | 107     | 67                       | 307   | 21                     | 12-7             | 12-7   |
| H                  | 110     | 70                       | 310   | 22                     | 12-8             | 12-8   |
| I                  | 111     | 71                       | 311   | 23                     | 12-9             | 12-9   |
| J                  | 112     | 41                       | 312   | 24                     | 11-1             | 11-1   |
| K                  | 113     | 42                       | 313   | 25                     | 11-2             | 11-2   |
| L                  | 114     | 43                       | 314   | 26                     | 11-3             | 11-3   |
| M                  | 115     | 44                       | 315   | 27                     | 11-4             | 11-4   |
| N                  | 116     | 45                       | 316   | 30                     | 11-5             | 11-5   |
| O                  | 117     | 46                       | 317   | 31                     | 11-6             | 11-6   |
| P                  | 120     | 47                       | 320   | 32                     | 11-7             | 11-7   |
| Q                  | 121     | 50                       | 321   | 33                     | 11-8             | 11-8   |
| R                  | 122     | 51                       | 322   | 34                     | 11-9             | 11-9   |
| S                  | 123     | 22                       | 323   | 35                     | 0-2              | 0-2    |

## APPENDIX D – STANDARD CHARACTER CODES

| Print<br>Character | Printer | Mag Tape<br>(7 trk. BCD) | ASCII | FORTRAN<br>(internal) | Hollerith<br>026 | 029    |
|--------------------|---------|--------------------------|-------|-----------------------|------------------|--------|
| T                  | 124     | 23                       | 324   | 36                    | 0-3              | 0-3    |
| U                  | 125     | 24                       | 325   | 37                    | 0-4              | 0-4    |
| V                  | 126     | 25                       | 326   | 40                    | 0-5              | 0-5    |
| W                  | 127     | 26                       | 327   | 41                    | 0-6              | 0-6    |
| X                  | 130     | 27                       | 330   | 42                    | 0-7              | 0-7    |
| Y                  | 131     | 30                       | 331   | 43                    | 0-8              | 0-8    |
| Z                  | 132     | 31                       | 332   | 44                    | 0-9              | 0-9    |
| [                  | 133     | 75                       | 333   | 76 <sup>2</sup>       | 12-5-8           | 12-2-8 |
| \                  | 134     | 36                       | 334   | 76 <sup>2</sup>       | 0-6-8            | 11-7-8 |
| ]                  | 135     | 55                       | 335   | 76 <sup>2</sup>       | 11-5-8           | 0-2-8  |
| ↑                  | 136     | 17 <sup>1</sup>          | 336   | 76 <sup>2</sup>       | 7-8              | 12-7-8 |
| ←                  | 137     | 20                       | 337   | 76 <sup>3</sup>       | 2-8              | 0-5-8  |

1. End-of-file for magnetic tape (7 track).
2. Undefined character for Stand-alone FORTRAN IV compiler and runtime.
3. Teletype form control character for all FORTRAN IV compilers and runtimes (skip to column 1).
4. Teletype form control character for Stand-alone FORTRAN IV compiler and runtime (skip to column 7).

## APPENDIX E – TTY CHARACTER CODES

| Character | 620/622 Internal Code | Character | 620/622 Internal Code |
|-----------|-----------------------|-----------|-----------------------|
| 0         | 260                   | Y         | 331                   |
| 1         | 261                   | Z         | 332                   |
| 2         | 262                   | (blank)   | 240                   |
| 3         | 263                   |           | 241                   |
| 4         | 264                   | ,         | 242                   |
| 5         | 265                   | #         | 243                   |
| 6         | 266                   | \$        | 244                   |
| 7         | 267                   |           | 245                   |
| 8         | 270                   | &         | 246                   |
| 9         | 271                   | '         | 247                   |
| A         | 301                   | (         | 250                   |
| B         | 302                   | )         | 251                   |
| C         | 303                   | *         | 252                   |
| D         | 304                   | +         | 253                   |
| E         | 305                   | ,         | 254                   |
| F         | 206                   | —         | 255                   |
| G         | 307                   | •         | 256                   |
| H         | 310                   | /         | 257                   |
| I         | 311                   | :         | 272                   |
| J         | 312                   | ;         | 273                   |
| K         | 313                   |           | 274                   |
| L         | 314                   | =         | 275                   |
| M         | 315                   |           | 276                   |
| N         | 316                   |           | 277                   |
| O         | 317                   | @         | 300                   |
| P         | 320                   |           | 333                   |
| Q         | 321                   |           | 334                   |
| R         | 322                   |           | 335                   |
| S         | 323                   |           | 336                   |
| T         | 324                   |           | 337                   |
| U         | 325                   | RUBOUT    | 377                   |
| V         | 326                   | NUL       | 200                   |
| W         | 327                   | SOM       | 201                   |
| X         | 330                   | EOA       | 202                   |

Teletypewriter Character Codes (1 of 2)

| Character | 620/622 Internal Code | Character | 620/622 Internal Code |
|-----------|-----------------------|-----------|-----------------------|
| EOM       | 203                   | X-OFF     | 223                   |
| EOT       | 204                   | TAPE OFF  |                       |
| WRU       | 205                   | AUX       | 224                   |
| RU        | 206                   | ERROR     | 225                   |
| BEL       | 207                   | SYNC      | 226                   |
| FE        | 210                   | LEM       | 227                   |
| H TAB     | 211                   | S0        | 230                   |
| LINE FEED | 212                   | S1        | 231                   |
| V TAB     | 213                   | S2        | 232                   |
| FORM      | 214                   | S3        | 233                   |
| RETURN    | 215                   | S4        | 234                   |
| SO        | 216                   | S5        | 235                   |
| SI        | 217                   | S6        | 236                   |
| DCO       | 220                   | S7        | 237                   |
| X-ON      | 221                   |           |                       |
| TAPE AUX  |                       |           |                       |
| ON        | 222                   |           |                       |





## NOTES

## NOTES

ADDENDUM 1  
SOFTWARE HANDBOOK  
Varian Document 98 A 9952 200

This addendum lists changes and supplementary information for the Software Handbook, page BLD 1-3, instruction a.

The following table gives the Bootstrap Loader Routines for BLD II in systems having no automatic bootstrap loader.

| Location | High-Speed Reader | Model-B<br>Teletype (620-06B) | Mnemonic            |
|----------|-------------------|-------------------------------|---------------------|
| 07756    | 102637            | 102601                        | CIB                 |
| 07757    | 004011*           | 004011*                       | ASLB                |
| 07760    | 004041            | 004041                        | LRLB                |
| 07761    | 004446            | 004446                        | LLRL                |
| 07762    | 001020            | 001020                        | JBZ                 |
| 07763    | 007772            | 007772                        | (memory<br>address) |
| 07764    | 055000            | 055000                        | STA                 |
| 07765    | 001010            | 001010                        | JAZ                 |
| 07766    | 007000            | 007000                        | (memory<br>address) |
| 07767    | 005144            | 005144                        | IXR                 |
| 07770    | 005101            | 005101                        | INCR                |
| 07771    | 100537            | 102601                        | CIB                 |
| 07772    | 101537            | 101201                        | SEN                 |
| 07773    | 007756            | 007756                        | (memory<br>address) |
| 07774    | 001000            | 001000                        | JMP                 |
| 07775    | 007772            | 007772                        | (memory<br>address) |

\* For 18-bit computers insert 004013.

issued NOVEMBER 1972



(please detach card here)

First Class  
Permit No. 323  
Newport Beach  
California

**BUSINESS REPLY MAIL** No Postage Stamp Necessary if Mailed in United States

postage will be paid by



**varian data machines**  
2722 michelson drive  
irvine/california/92664



I AM INTERESTED IN:

- ☐ Varian 620/f Computer
- ☐ Varian 620/L Computer
- ☐ Varian R 620 Ruggedized Computer
- ☐ Varian 620/DC Data Communication System

Application:

- \_\_\_\_ Batch processing
- \_\_\_\_ Analytical instrumentation data
- \_\_\_\_ Telemetry or aerospace test data
- \_\_\_\_ Small computer used as a local satellite to a larger computer for control of I/O peripherals
- \_\_\_\_ Small computer used as a remote satellite to a larger computer for control of I/O peripherals
- \_\_\_\_ Numerical control of machine tools or drafting machines
- \_\_\_\_ Control of chemical, petroleum, or power generation processes
- \_\_\_\_ Communication switching, data concentration, etc.
- \_\_\_\_ Production line testing
- \_\_\_\_ Other \_\_\_\_\_

(PLEASE SPECIFY)

- ☐ Please have a sales engineer call me
- ☐ Please add my name to your mailing list

*Mail to:*

NAME \_\_\_\_\_

TITLE \_\_\_\_\_

COMPANY \_\_\_\_\_ DEPT. CODE \_\_\_\_\_

ADDRESS \_\_\_\_\_

CITY \_\_\_\_\_ PHONE \_\_\_\_\_

STATE \_\_\_\_\_ ZIP \_\_\_\_\_

For VDM Use Only





\$3.00

**varian**  
**software**  
**handbook**



**varian data machines**

2722 michelson drive  
irvine/california/92664